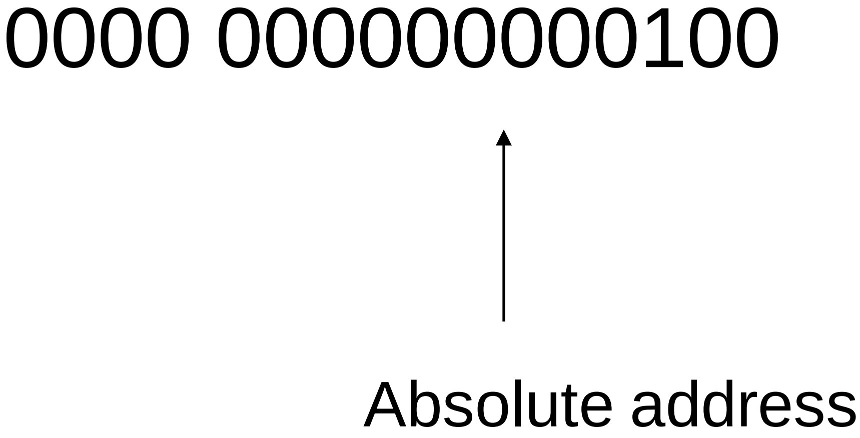
# Chapter 4

H1 Assembly Language: Part 2

# Direct instruction

Contains the absolute address of the memory location it accesses.

ld instruction:



0000 000000000100

Absolute address

Shorthand notation for ld    x

```
    ac = mem[x];
```

where

$0 <= x <= 4095$

## 4.3 DIRECT INSTRUCTIONS

| Opcode | Assembly | | |
|---|---|---|---|
| (hex) | Form | Name | Description |
| 0 | ld x | Load | ac = mem[x]; |
| 1 | st x | Store | mem[x] = ac; |
| 2 | add x | Add | ac = ac + mem[x]; |
| 3 | sub x | Subtract | ac = ac - mem[x]; |

# Stack instructions

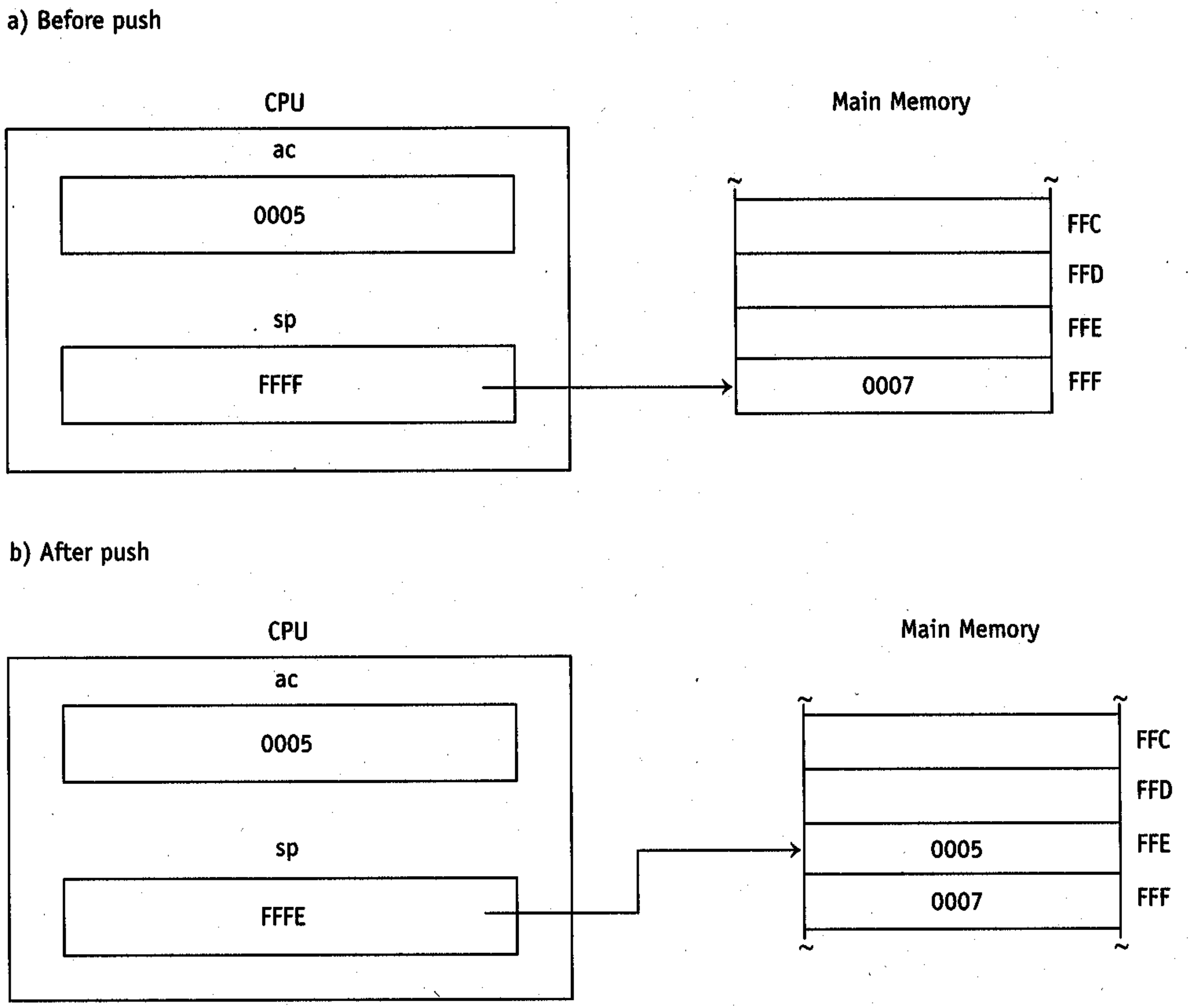
- **push** pushes the ac register contents onto the top of the stack.
- **pop** removes the value of top of the stack and loads it into the ac register.
- **swap** exchanges the values in the ac and sp registers.
- sp register points to the top of the stack.
- sp is pre-decremented on a push.

# 4.4 STACK INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| F3 | push | Push onto stack | mem[--sp] = ac; |
| F4 | pop | Pop from stack | ac = mem[sp++]; |
| F7 | swap | Swap | temp = ac; ac = sp; sp = temp; |

temp is a work register within the CPU.

FIGURE 4.1



a) Before push

b) After push

# Immediate instructions

An *immediate instruction* contains the operand—not the operand address as in the direct instructions.  Because it is in the instruction, the operand is “immediately” available.

## 4.5 IMMEDIATE INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| 8 | ldc x | Load constant | ac = x; |
| F5 | aloc y | Allocate | sp = sp - y; |
| F6 | dloc y | Deallocate | sp = sp + y; |

0 ≤ x ≤ FFF hex = 4095 decimal

0 ≤ y ≤ FF hex = 255 decimal

# ldc 1

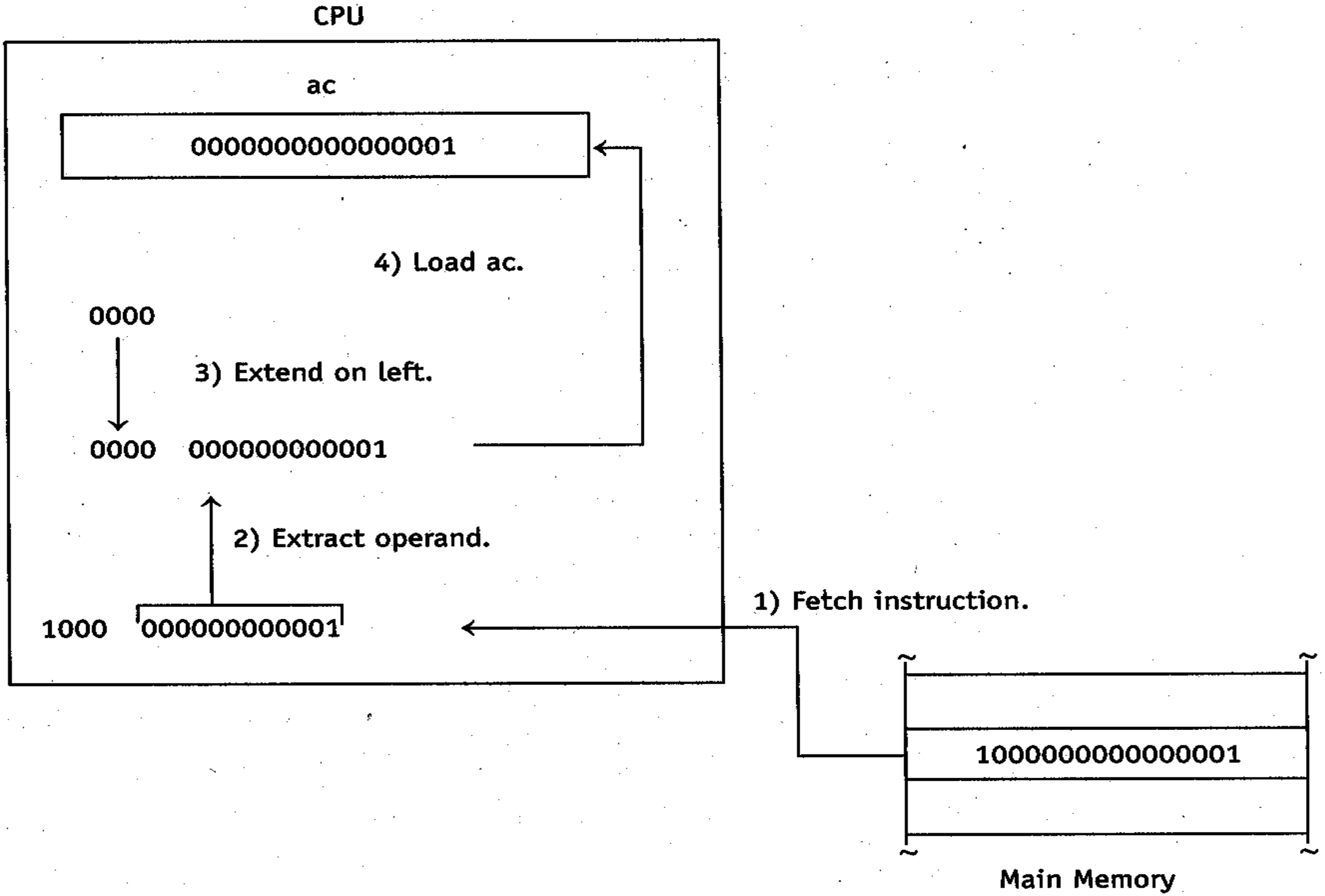
Machine code:

1000 000000000001

Loads 1 (the operand in the instruction itself) into the ac register (zero extends operand to 16 bits).

# Execution of ldc 1

FIGURE 4.2



# What does this program do?

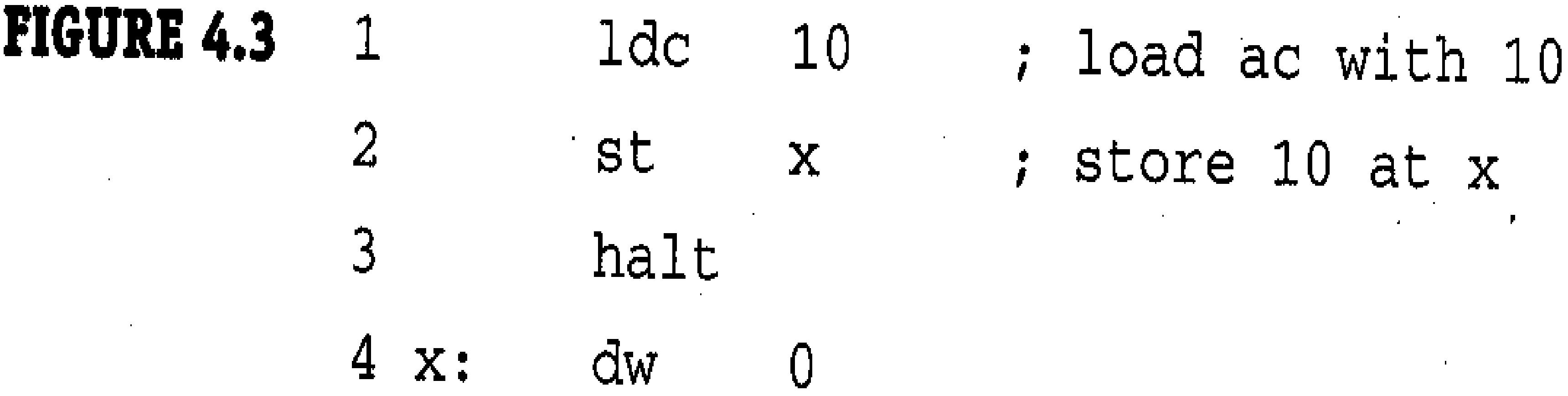


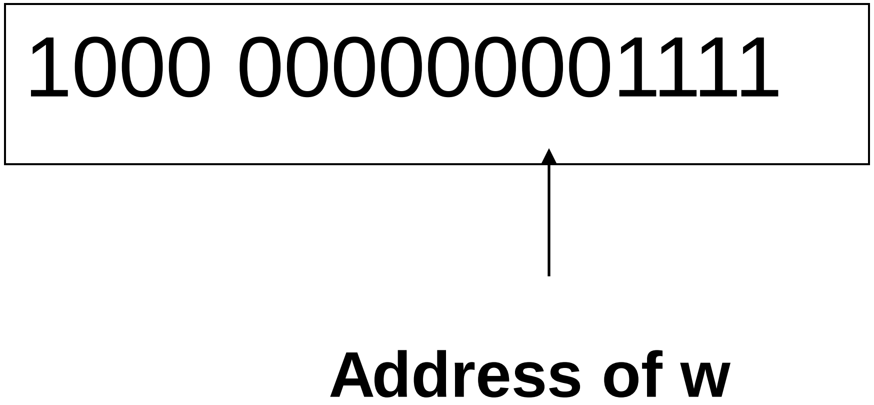
FIGURE 4.3

```
        ldc   10      ; load ac with 10
        st    x       ; store 10 at x
        halt
x:      dw    0
```

# What an assembler does

- Translates mnemonics to binary opcodes.
- Translate labels to binary addresses.
- Translates numbers to binary.
- Translates strings to ASCII codes.

```
ld    x      0000 0000 0000 0100

ldc   x      1000 0000 0000 0100

ldc   5      1000 0000 0000 0101

dw    ‘A’    00000000 01000001
```



1000 000000001111

**Address of w**

ldc  w

This instruction loads the **address** of w into the ac register.

FIGURE 4.4

```
          ld    w       ; loads 2
          st    x
          ldc   w       ; loads 7, the address of w
          st    y
          ldc   1       ; loads the constant 1
          st    z
          halt
w:        dw    2
x:        dw    0
y:        dw    0
z:        dw    0
```

ldc 'A'

1000 000001000001

↑

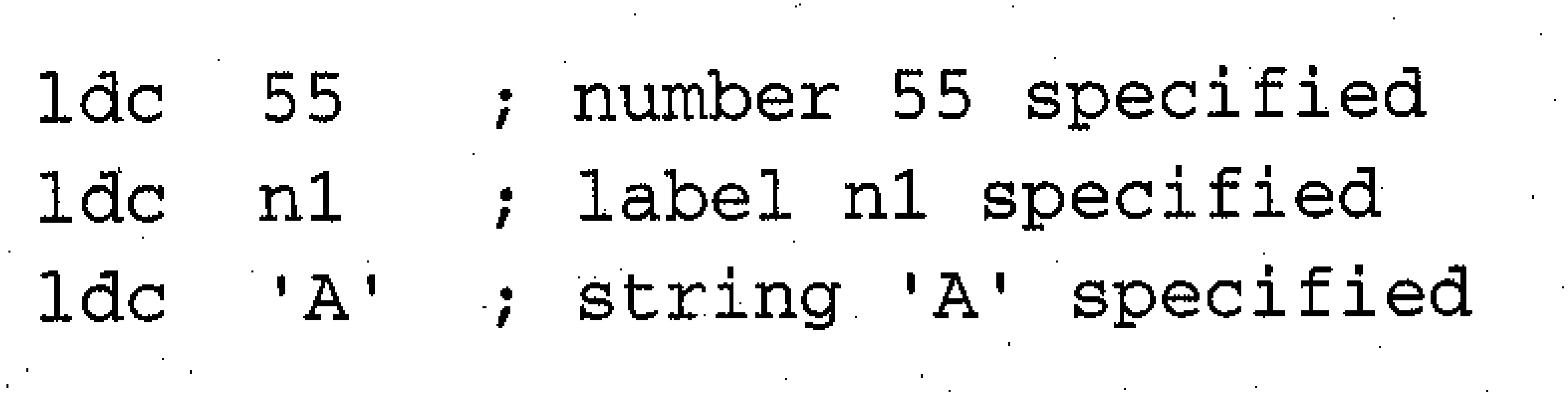
ASCII code for 'A'

This instruction loads the ASCII code for 'A' into the ac register.

**FIGURE 4.5**

```
LOC OBJ  SOURCE
hex*dec
0   *0   8041      ldc    'A'    ; immediate operand is 041
1   *1   1007      st      x     ; store code in ac to x
2   *2   8042      ldc    'B'    ; immediate operand is 042
3   *3   1008      st      y     ; store code in ac to y
4   *4   8043      ldc    'C'    ; immediate operand is 043
5   *5   1009      st      z     ; store code in ac to z
6   *6   FFFF      halt
7   *7   0000      x:     dw     0
8   *8   0000      y:     dw     0
9   *9   0000      z:     dw     0
A   *10  ========= end of fig0405.mas ============================
```

# Uses of the ldc instruction



```
ldc  55    ; number 55 specified
ldc  n1    ; label n1 specified
ldc  'A'   ; string 'A' specified
```

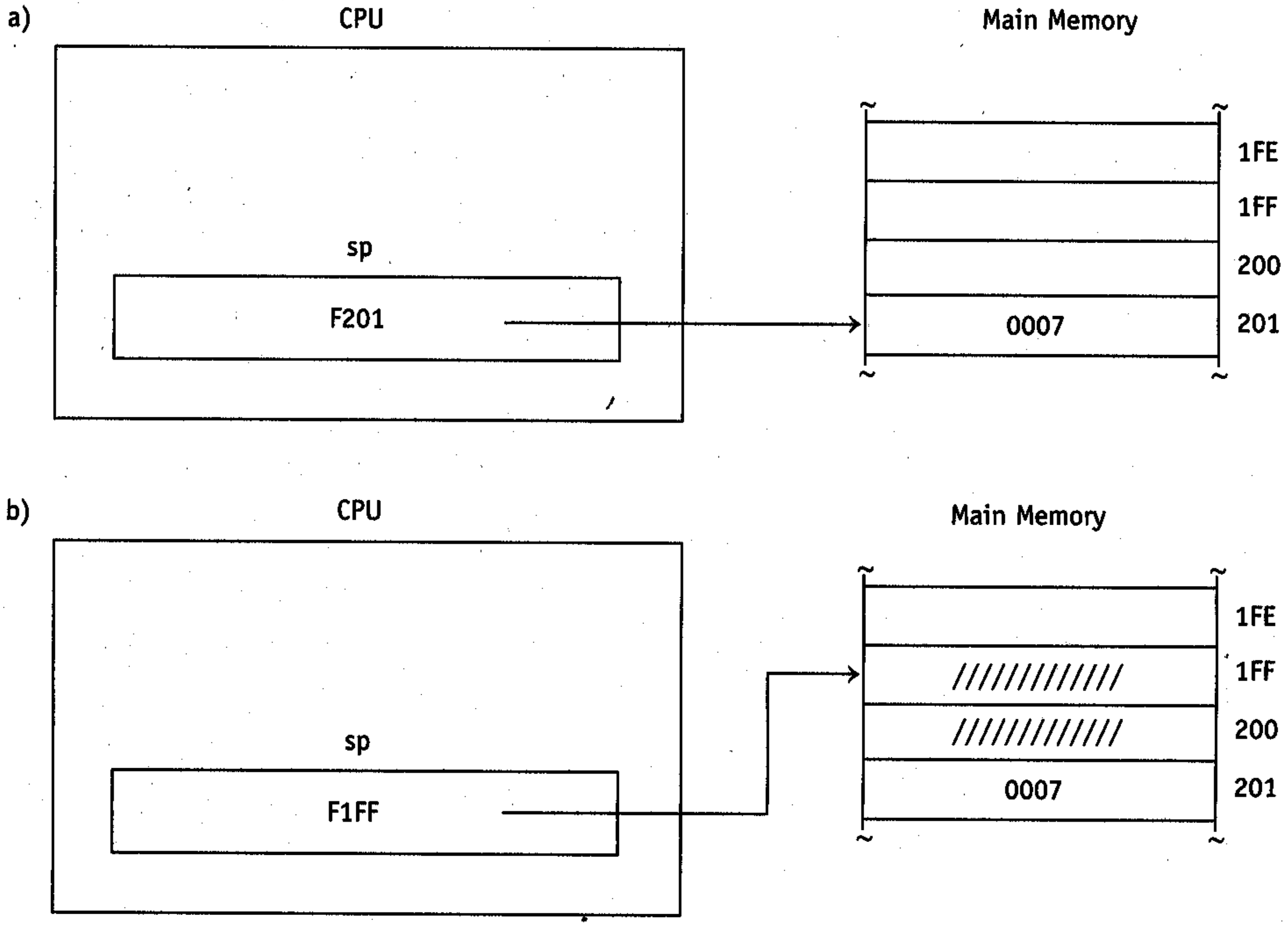
# aloc and dloc instructions

- alloc 2 subtracts 2 from sp register, reserving two slots on the stack

- dloc 2 adds 2 to the sp register, deallocating two slots on the stack.

# Effect of aloc 2

FIGURE 4.6



# 4.6 I/O INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FFF5 | uout | Unsigned output | Output number in ac as unsigned decimal number |
| FFF6 | sin | String input | Input string to address in ac |
| FFF7 | sout | String output | Output string pointed to by ac |
| FFF8 | hin | Hex input | Input hex number to ac |
| FFF9 | hout | Hex output | Output number in ac in hex |
| FFFA | ain | ASCII input | Input ASCII char to ac |
| FFFB | aout | ASCII output | Output ASCII char in ac |
| FFFC | din | Decimal input | Input decimal number (signed or unsigned) to ac |
| FFFD | dout | Decimal output | Output number in ac as signed decimal number |

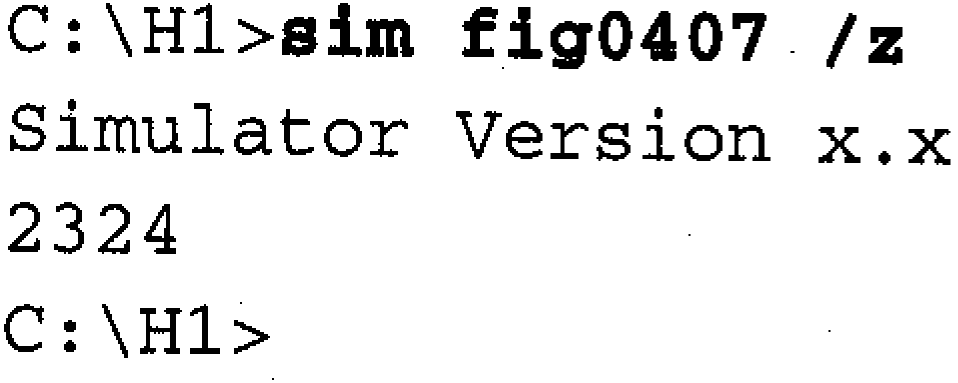
**FIGURE 4.7**

```
        ldc     23      ; load ac with 23
        dout            ; output 23 to display
        add     @1      ; add 1 to ac
        dout            ; output 24 to display
        halt
  @1:   dw      1
```

**FIGURE 4.8**

```
Starting session.  Enter h or ? for help.
---- [T7] 0: ldc  /8 017/ g        ←go to halt
   0: ldc  /8 017/ ac=0000/0017
   1: dout /FFFD / 23              ←output from dout
   2: add  /2 005/ ac=0017/0018
   3: dout /FFFD / 24              ←output from dout
   4: halt /FFFF /
Machine inst count =       5 (hex) =     5 (dec)
---- [T7] g
Now at halt. Enter o to do over, q to quit, or h or ? for help.
---- [T7] o                        ←do over
Starting session.  Enter h or ? for help.
---- [T7] 0: ldc  /8 017/ n        ←no display
No display mode
---- [T1] g
2324                               ←output from dout instructions
Machine inst count =       5 (hex) =     5 (dec)
---- [T1] q
```

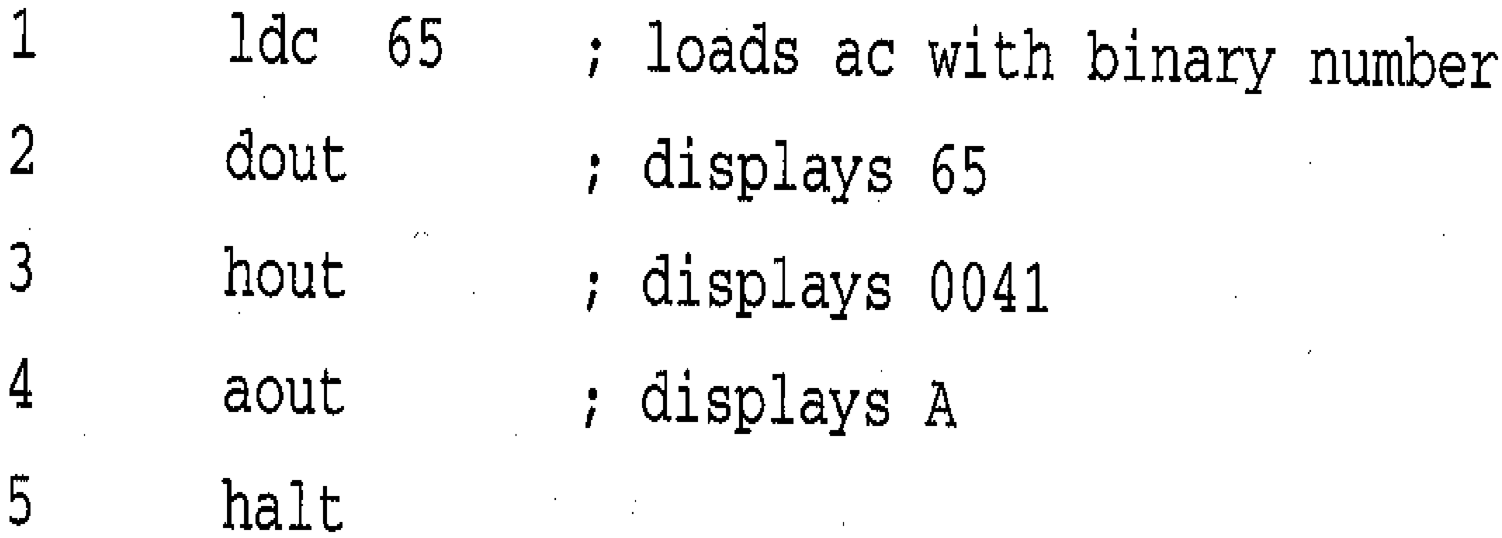
# Running sim without the debugger



```
C:\H1>sim fig0407 /z
Simulator Version x.x
2324
C:\H1>
```

# dout, hout, aout do not output newlines

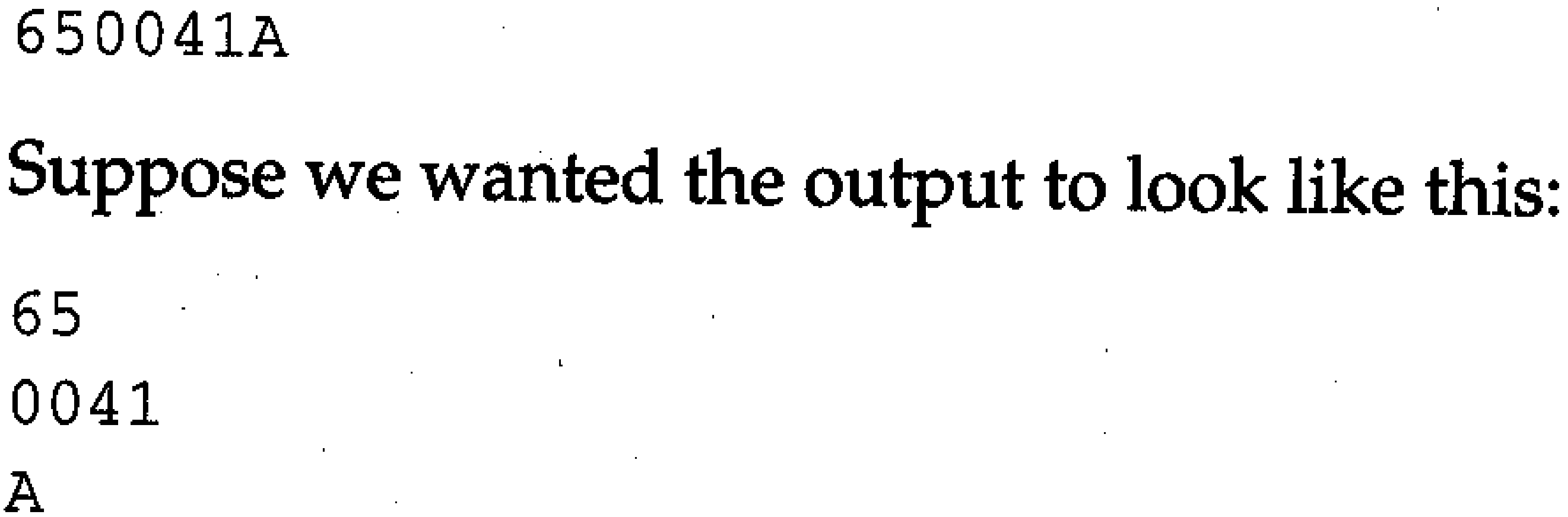
FIGURE 4.9



```
ldc  65         ; loads ac with binary number
dout            ; displays 65
hout            ; displays 0041
aout            ; displays A
halt
```

Output:   650041A

# Output of previous program is



```
650041A
```

Suppose we wanted the output to look like this:

```
65
0041
A
```

To output a newline character, use the aout instruction:

```
ldc  ‘\n’
aout
```

FIGURE 4.10

```
ldc   65     ; loads ac with binary number 0000000001000001
dout         ; displays 65
ldc   '\n'   ; load newline character
aout         ; output newline—that is go to next line
ldc   65     ; restore ac with 65
hout         ; displays 41
ldc   '\n'   ; load newline character
aout         ; output newline—that is go to next line
ldc   65     ; restore ac with 65
aout         ; displays A
ldc   '\n'   ; load newline character
aout         ; output newline—that is go to next line
halt
```

When an input instruction is executed, the system waits until the user enters the required input on the keyboard.

FIGURE 4.11

```
din     ; input decimal number
dout    ; output same decimal number
ldc '\n'
aout    ; go to next line
hin     ; input hex number
dout    ; output decimal equivalent
ldc '\n'
aout    ; go to next line
ain     ; input ASCII character
dout    ; output ASCII code in decimal
ldc '\n'
aout    ; go to next line
halt
```

If we run the program in Figure 4.11 and enter "12," "24," and "A," the screen will look like this (the user's inputs are in boldface):

```
12          (decimal 12 is read in)
12          (decimal 12 is echoed)
24          (hex 24 is read in)
36          (decimal equivalent of hex 24 is echoed)
A           ('A' is read in)
65          (ASCII code for 'A' in decimal is echoed)
```

# sin and sout

- sin and sout, respectively, input and output to the memory location pointed to by the ac register.
- sout continues until it reaches a null character.
- Be sure to use double-quoted strings with sout (because they are null terminated).

**FIGURE 4.12**

```
        ldc  inbuf     ; get address of input buffer
        sin            ; read string into inbuf
        sout           ; output string from inbuf
        ldc  '\n'      ; get newline character
        aout           ; go to next line
        ldc  msg1      ; get address of msg1
        sout           ; output msg1
        ld   x         ; load value of x
        dout           ; output value of x in decimal
        ldc  msg2      ; get address of msg2
        sout           ; output msg2
        halt
x:      dw   5
inbuf:  dw   81 dup 0
msg1:   dw   "x = "
msg2:   dw   "(decimal)\n"
```

```
hello, world
hello, world
x = 5 (decimal)
```

## 4.7 JUMP INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| 9 | ja x | Jump always | pc = x; |
| A | jzop x | Jump zero or pos | if (ac ≥ 0) pc = x; |
| B | jn x | Jump negative | if (ac < 0) pc = x; |
| C | jz x | Jump zero | if (ac == 0) pc = x; |
| D | jnz x | Jump nonzero | if (ac != 0) pc = x; |

# How does a jump instruction work?

```
                 Machine code
   ja    1          9001
```

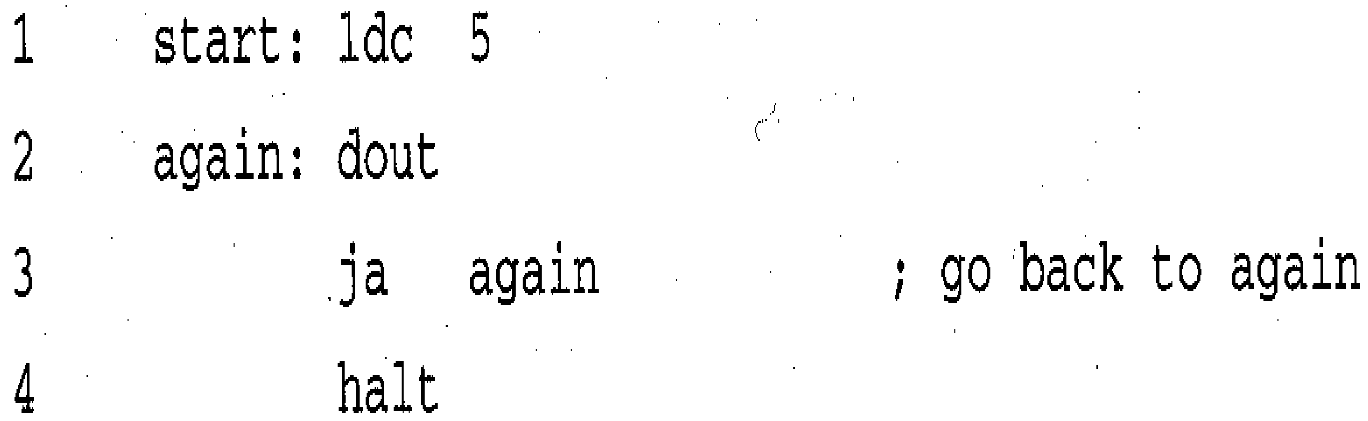
The execution of this instruction loads 1 into the pc register.

# Conditional Jump Instructions

```
ldc   5
jz    xxx    ; no jump

ldc   5
jnz   yyy    ; jump occurs
```

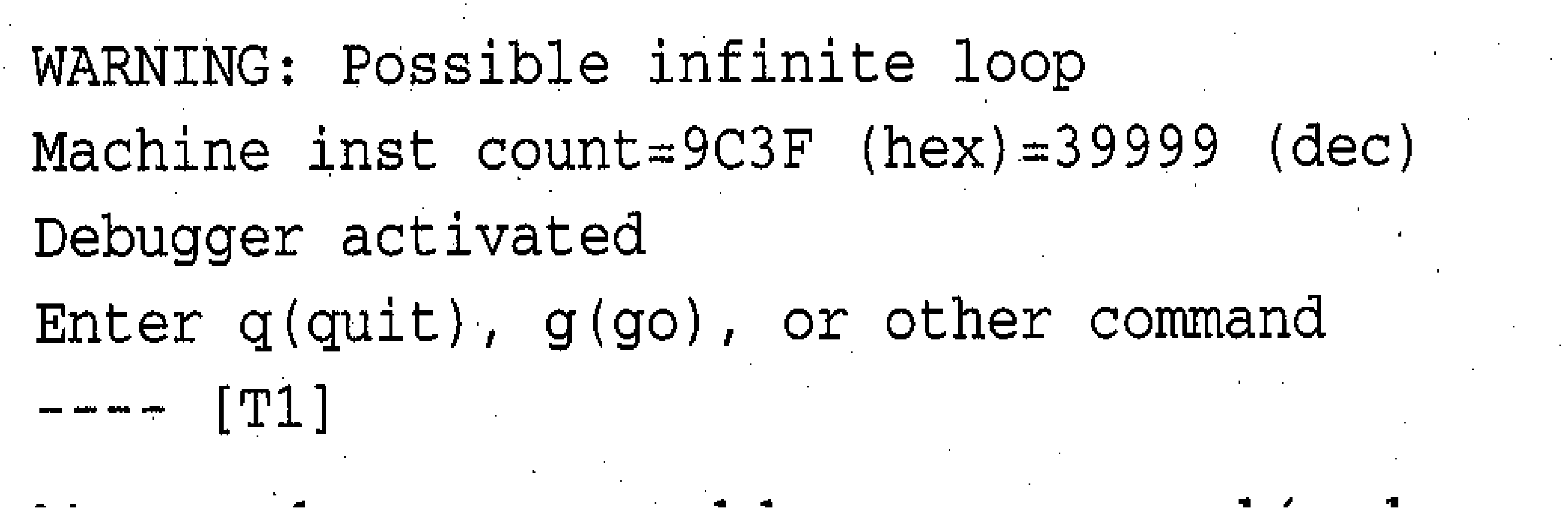
# Infinite loop

FIGURE 4.13



```
start: ldc   5
again: dout
       ja    again            ; go back to again
       halt
```

# Sim’s response to infinite loop



```
WARNING: Possible infinite loop
Machine inst count=9C3F (hex)=39999 (dec)
Debugger activated
Enter q(quit), g(go), or other command
---- [T1]
```

# Count-controlled loop

A loop whose number of iterations depends on a counter.

The program on the next slide computes 20 + 19 + …+ 1 using a count-controlled loop..

**FIGURE 4.14**

```
loop:   ld      sum         ; get sum
        add     count       ; add count to sum
        st      sum         ; store new sum
        ld      count       ; decrement count
        sub     @1
        st      count       ; put new value in count
        jnz     loop        ; repeat if count not zero
done:   ldc     msg         ; output "Sum = "
        sout
        ld      sum         ; output sum
        dout
        ldc     '\n'        ; output newline
        aout
        halt
@1:     dw      1
count:  dw      20
msg:    dw      "Sum = "
sum:    dw      0
```

# Indirect instruction

Accesses the memory location pointed to by the ac register.

Used to dereference pointers.

# 4.8 INDIRECT INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| F1 | `ldi` | Load indirect | ac = mem[ac]; |
| F2 | sti | Store indirect | mem[ac] = mem[sp++]; |

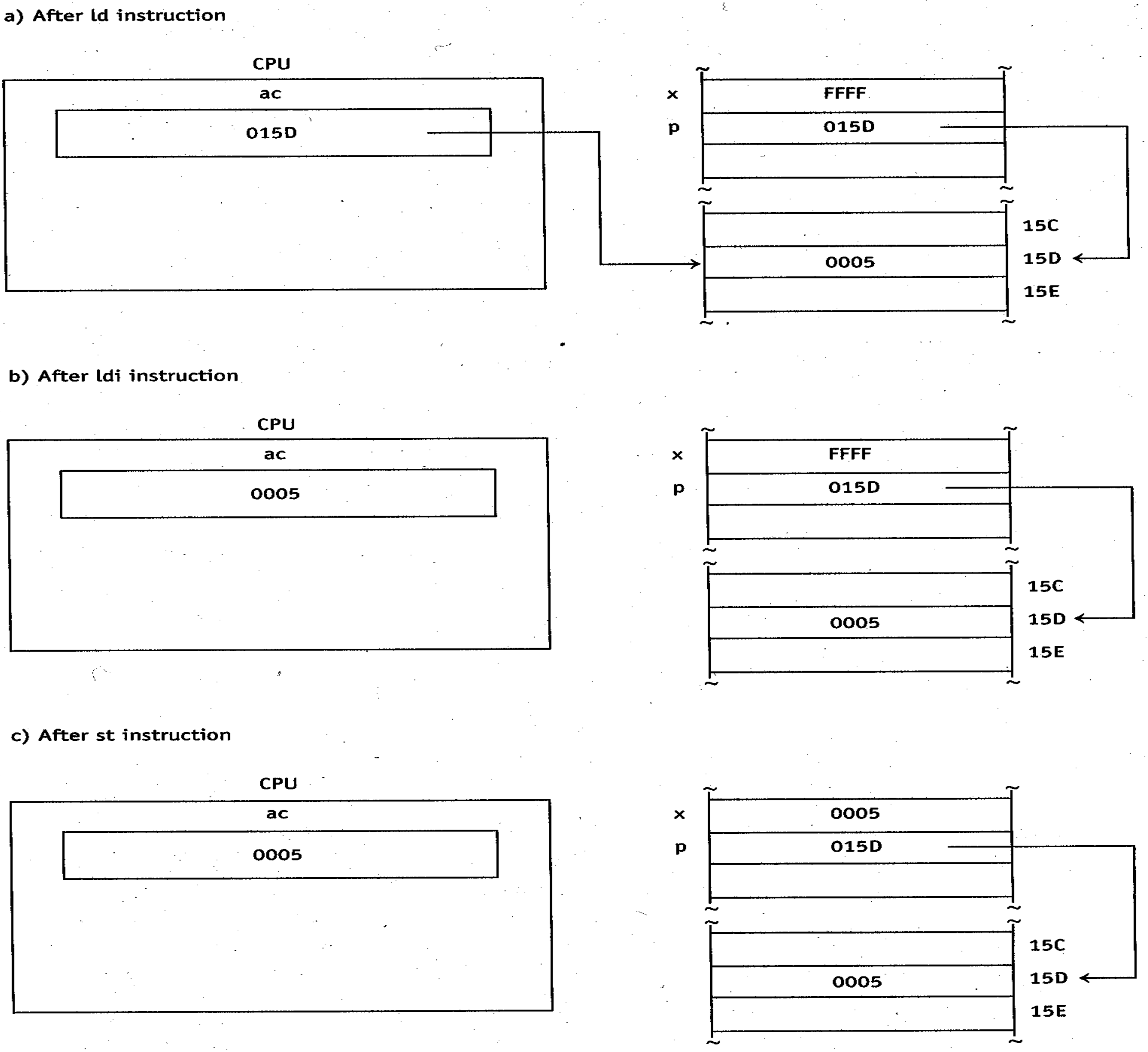
# ldi and sti instructions

- ldi loads the ac register from the memory location pointed to by the ac register.
- sti pops the stack and stores the popped value at the memory location pointed to by the ac register.

# Assembler code for
# x = *p;

```
ld   p        ; p contains an address
ldi
st   x
```

**FIGURE 4.15**



a) After ld instruction

b) After ldi instruction

c) After st instruction

# Assembler code for *p = 5;

```
ldc  5
push
ld  p
sti
```

FIGURE 4.16 a) After push and ld

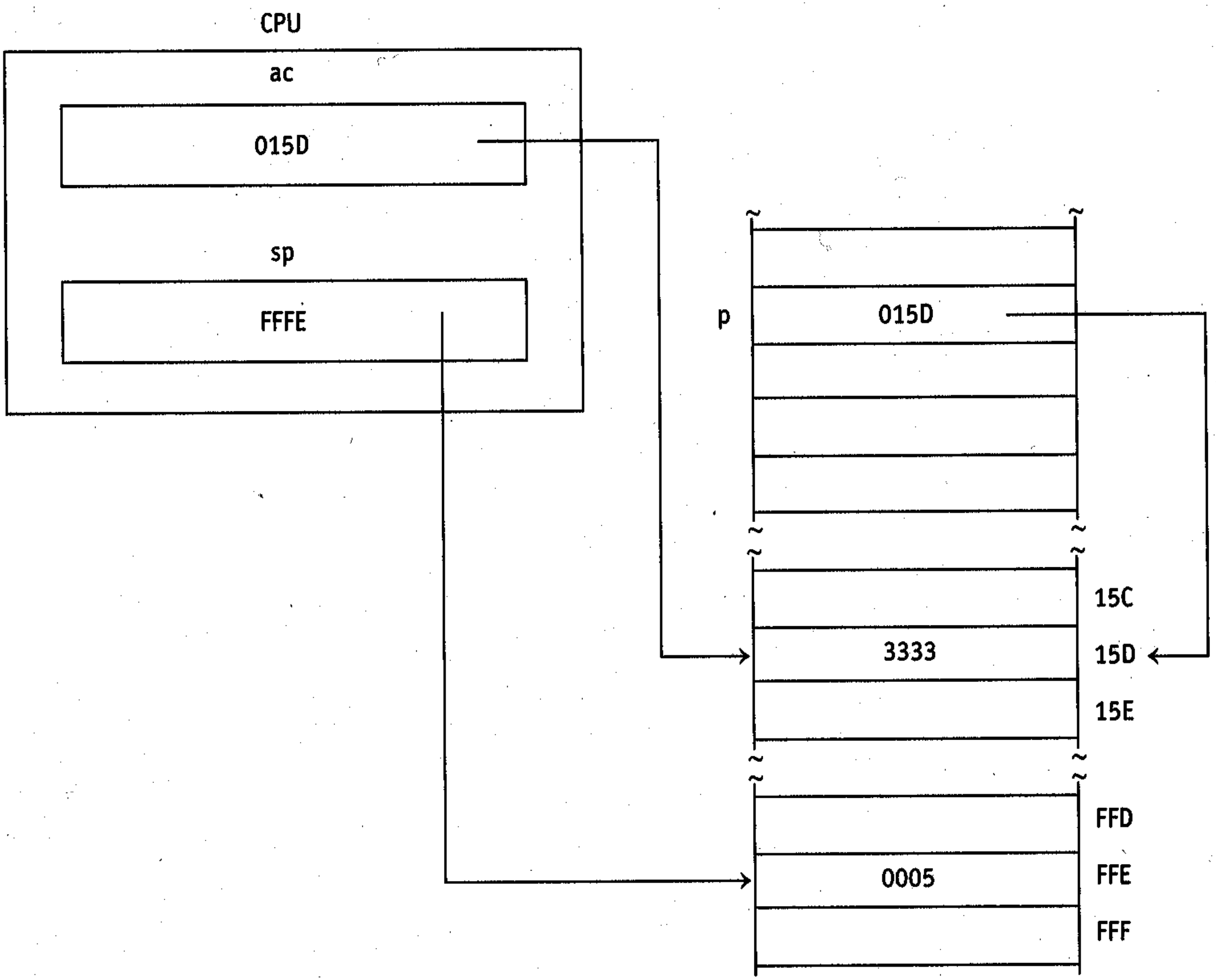
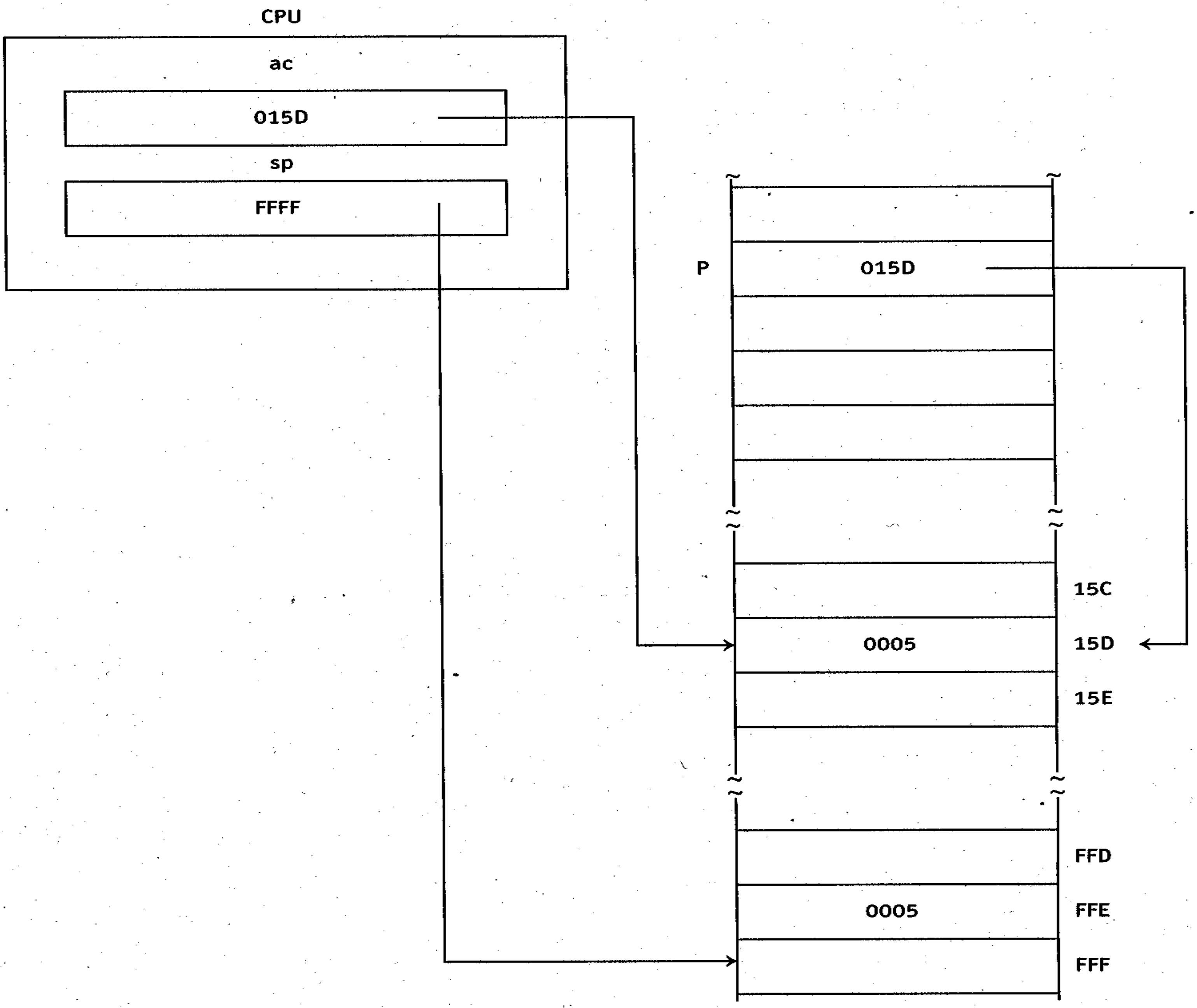


FIGURE 4.16 b) After sti instruction



A ***direct instruction*** holds an absolute address (i.e., its rightmost 12 bits).

An ***immediate instruction*** holds an operand.

A ***relative instruction*** holds a relative address.

A ***relative address*** is an address relative to the location to which sp points.

# 4.9 RELATIVE INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| 4 | ldr x | Load relative | ac = mem[sp + x]; |
| 5 | str x | Store relative | mem[sp + x] = ac; |
| 6 | addr x | Add relative | ac = ac + mem[sp + x]; |
| 7 | subr x | Subtract relative | ac = ac - mem[sp + x]; |

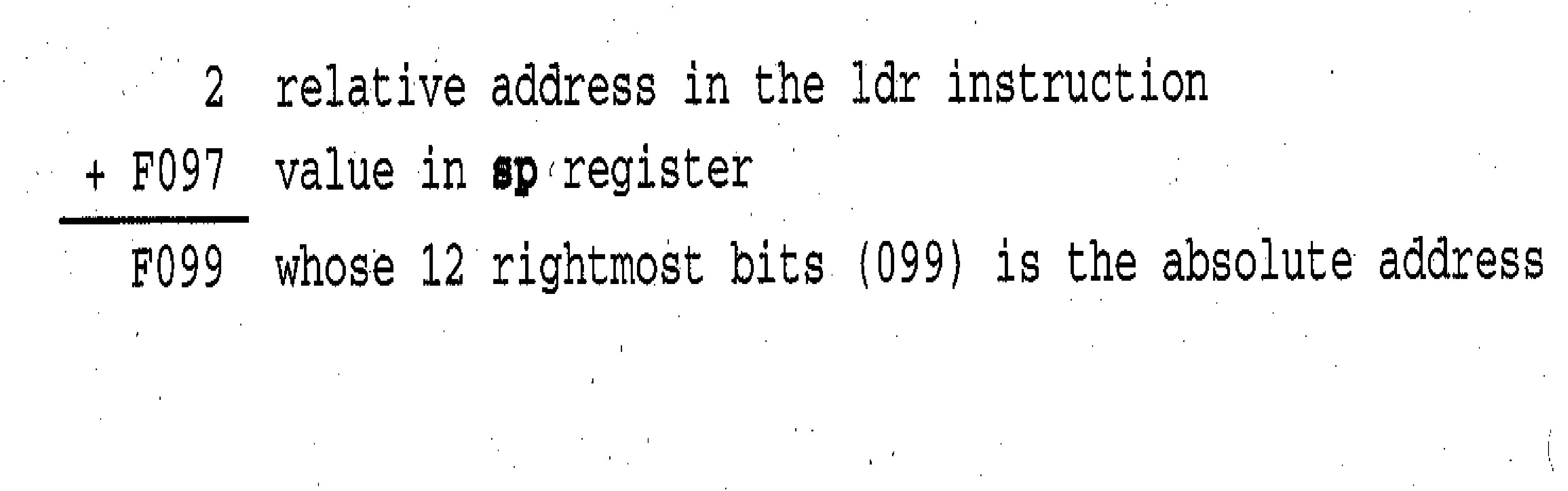
# What do these instructions do?

| | Machine code |
|---|---|
| ld 2 | 0002 |
| ldc 2 | 8002 |
| ldr 2 | 4002 |

x field

# Assume sp contains F097

ldr 2

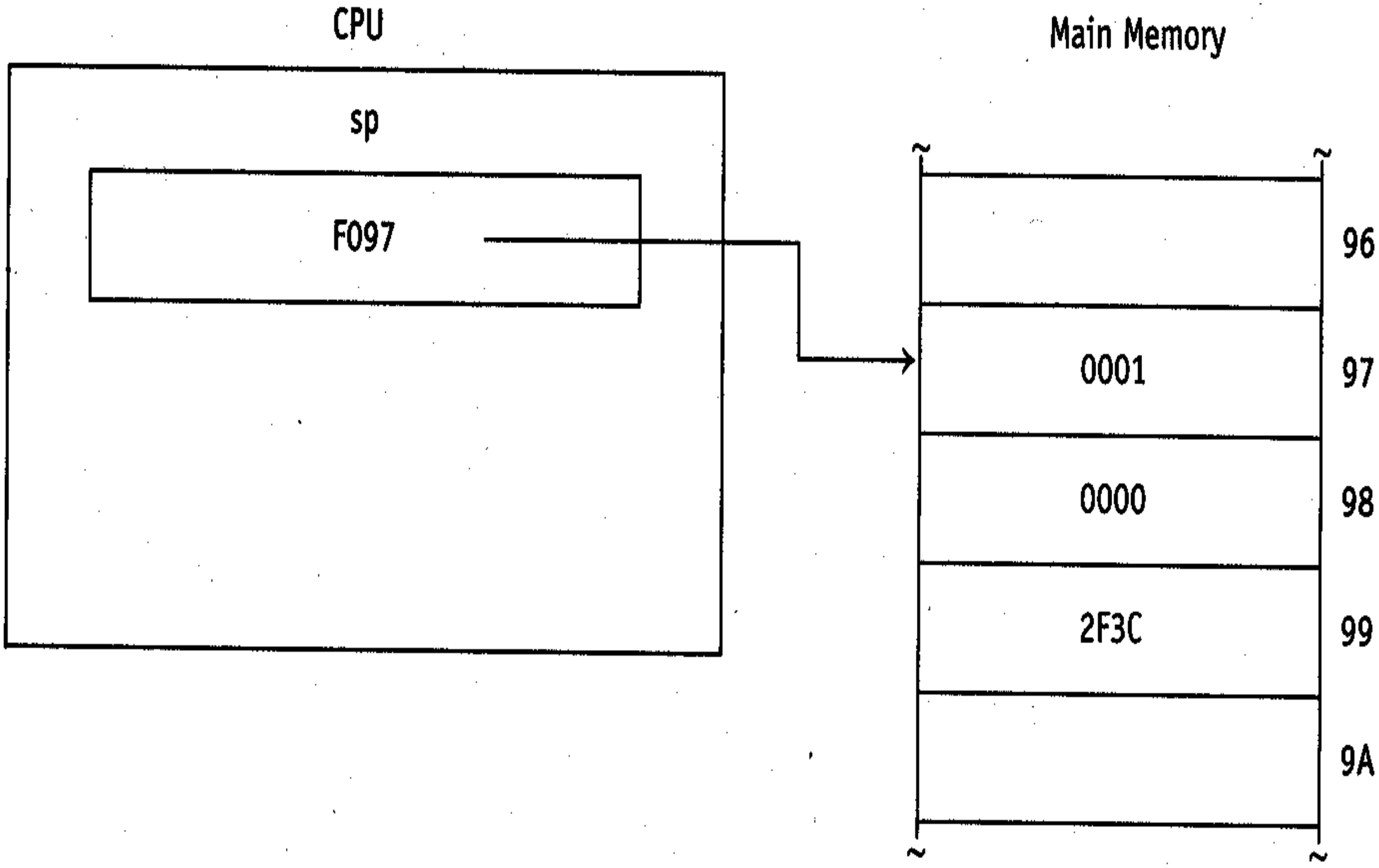


```
     2   relative address in the ldr instruction
 + F097  value in sp register
 ------
   F099  whose 12 rightmost bits (099) is the absolute address
```

Absolute address is 099

# ldr  2 loads what?

FIGURE 4.17



# Index register

- Used to access arrays in main memory.
- H1 does not have a dedicated index register.
- The sp register can be used as an index register.
- But sp is normally not available—it is usually needed as the top-of-stack pointer.

The program on next slide sums the numbers in the array **table** using sp as an index register.

**FIGURE 4.18**

```
             ldc    table
             swap             ;init sp with address of table
loop:        ld     sum
             addr   0         ; add number pointed to by sp
             st     sum
             dloc   1         ; move sp to next number in table
             ld     count
             sub    @1        ; decrement counter
             st     count
             jnz    loop      ; jump if counter not zero
             ldc    message   ; display sum
             sout
             ld     sum
             dout
             ldc    '\n'
             aout
             halt
message:     dw               "sum = "
@1:          dw               1
count:       dw               10
sum:         dw               0
table:       dw               56
             dw               -8
             dw               444
             dw               23
             dw               -233
             dw               16
             dw               45
             dw               -11
             dw               5
             dw               7
```

In place of the sp register, we can use a variable in memory as an index.  See the next slide.

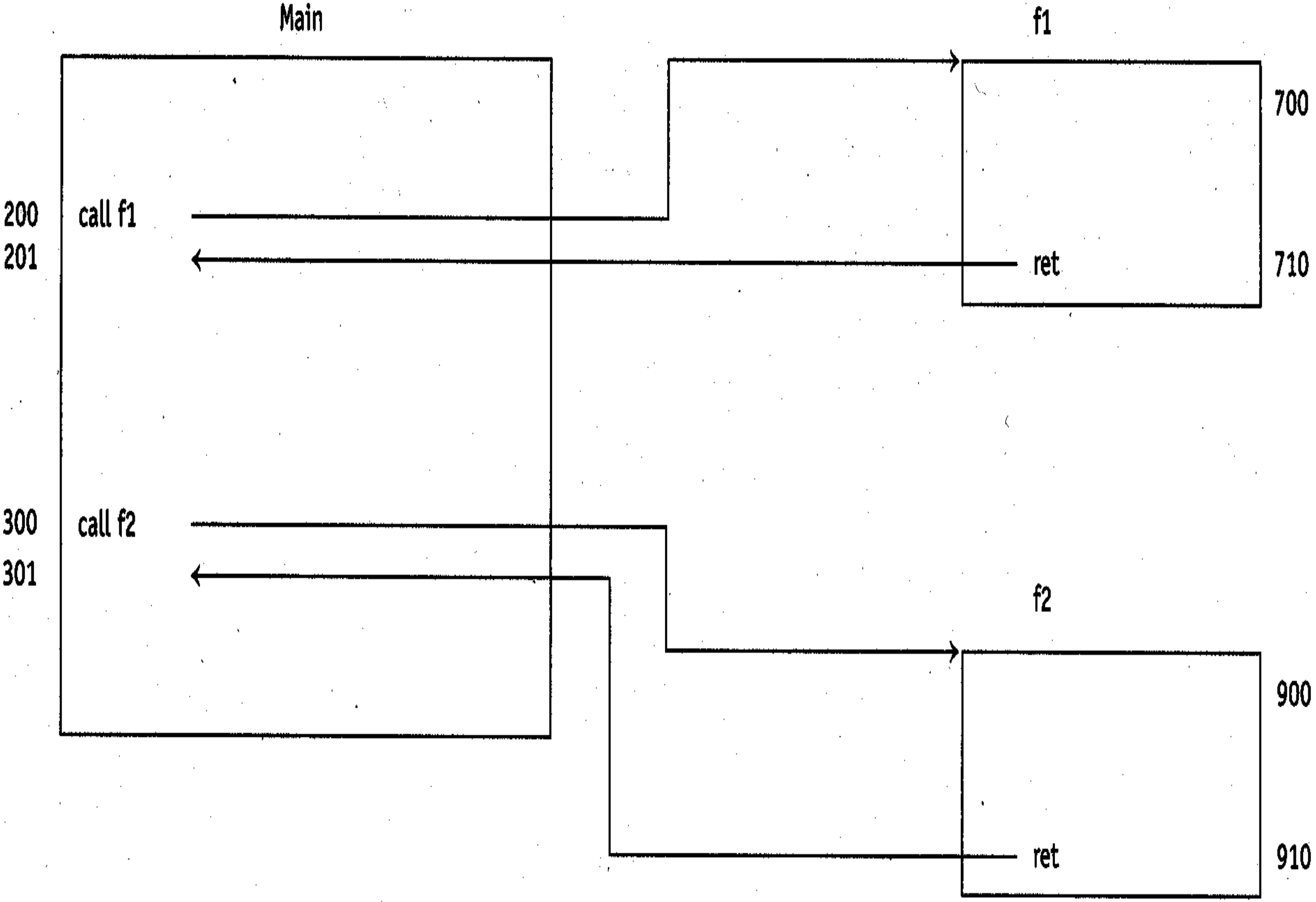
**FIGURE 4.19**

```
loop:     ldc    table      ; get address of table
          add    index      ; get address of table[index]
          ldi               ; load table[index]
          add    sum        ; add sum and table[index]
          st     sum        ; store result back in sum
          ld     index
          add    @1         ; increment index
          st     index
          ld     count
          sub    @1         ; decrement count
          st     count
          jnz    loop       ; jump if counter not zero
          ldc    message    ; display sum
          sout
          ld     sum
          dout
          ldc    '\n'
          aout
          halt
message:  dw     "sum = "
@1:       dw     1
count:    dw     10
sum:      dw     0
index:    dw     0
table:    dw     56
          dw     -8
          dw     444
          dw     23
          dw     -233
          dw     16
          dw     45
          dw     -11
          dw     5
          dw     7
```

# 4.11 LINKAGE INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| E | call x | Call procedure | mem[--sp] = pc; pc = x; |
| F0 | ret | Return | pc = mem[sp++]; |

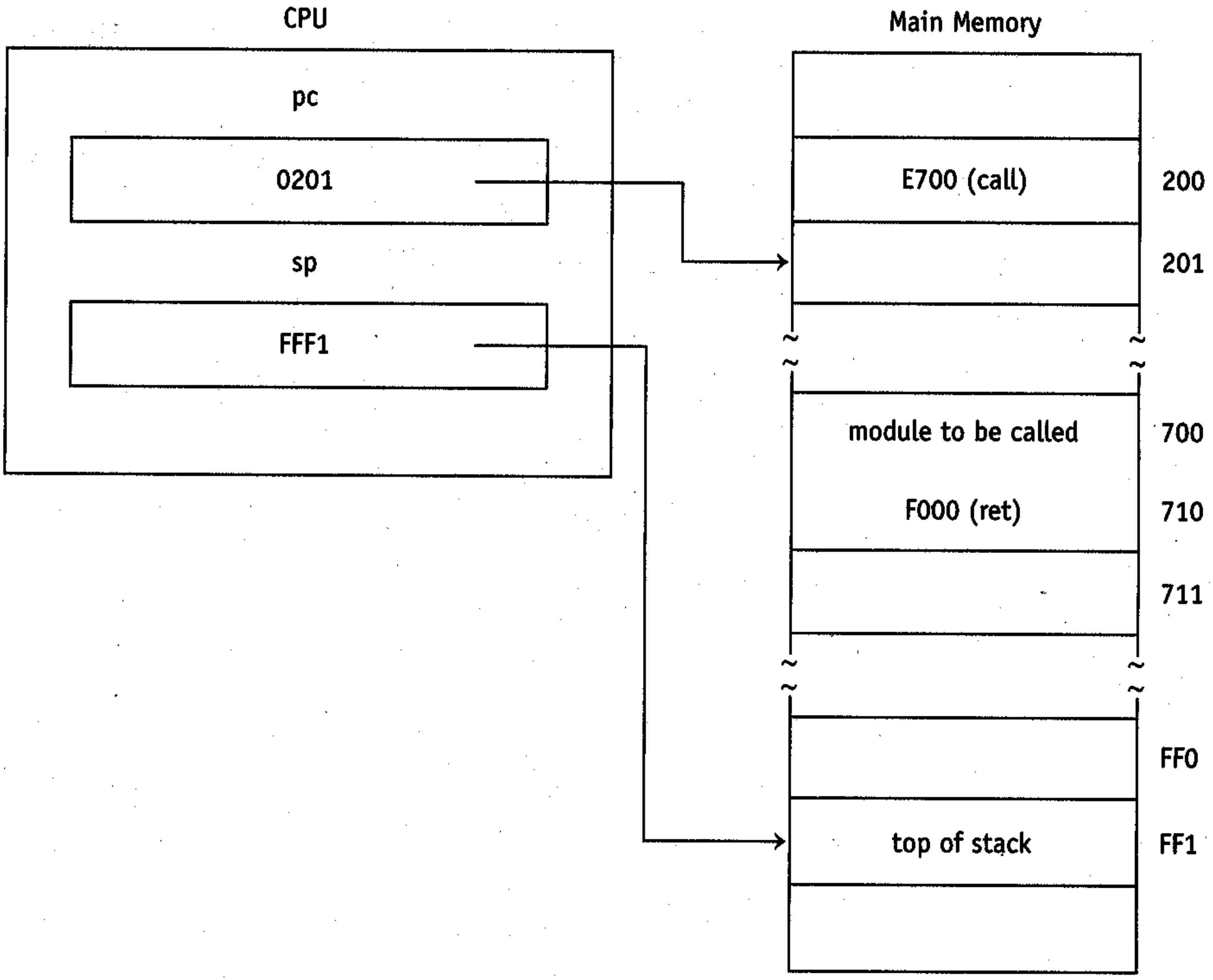
FIGURE 4.20


Main
f1
200
call f1
201
700
ret
710
300
call f2
301
f2
900
ret
910

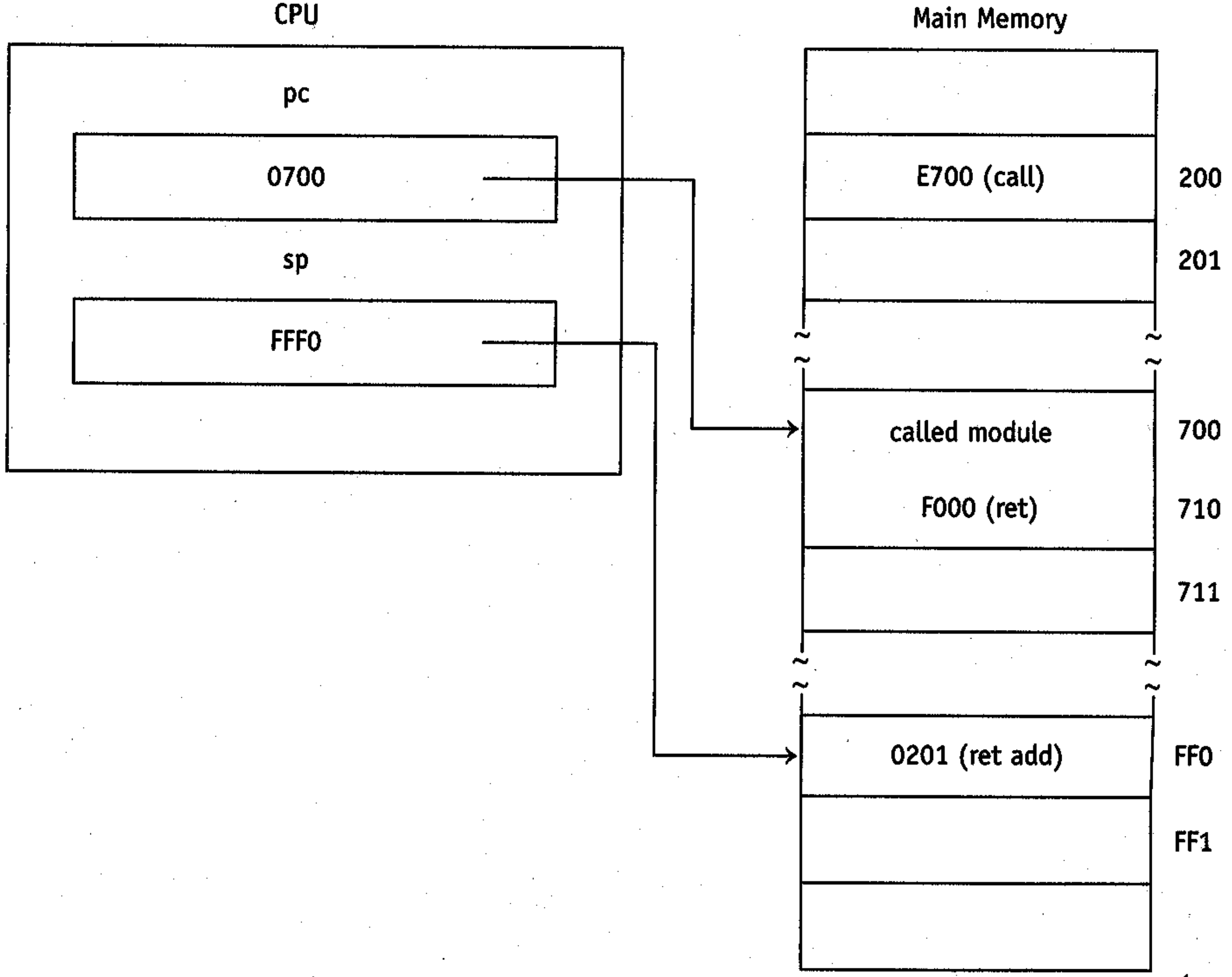
# call and ret instructions

- The call instruction saves the return address by pushing it onto the top of the stack (it pushes the pc register).
- The return instruction pops the top of the stack (which contains the return address) into the pc register.

**FIGURE 4.21** a) About to execute call instruction (step 4 in CPU cycle).

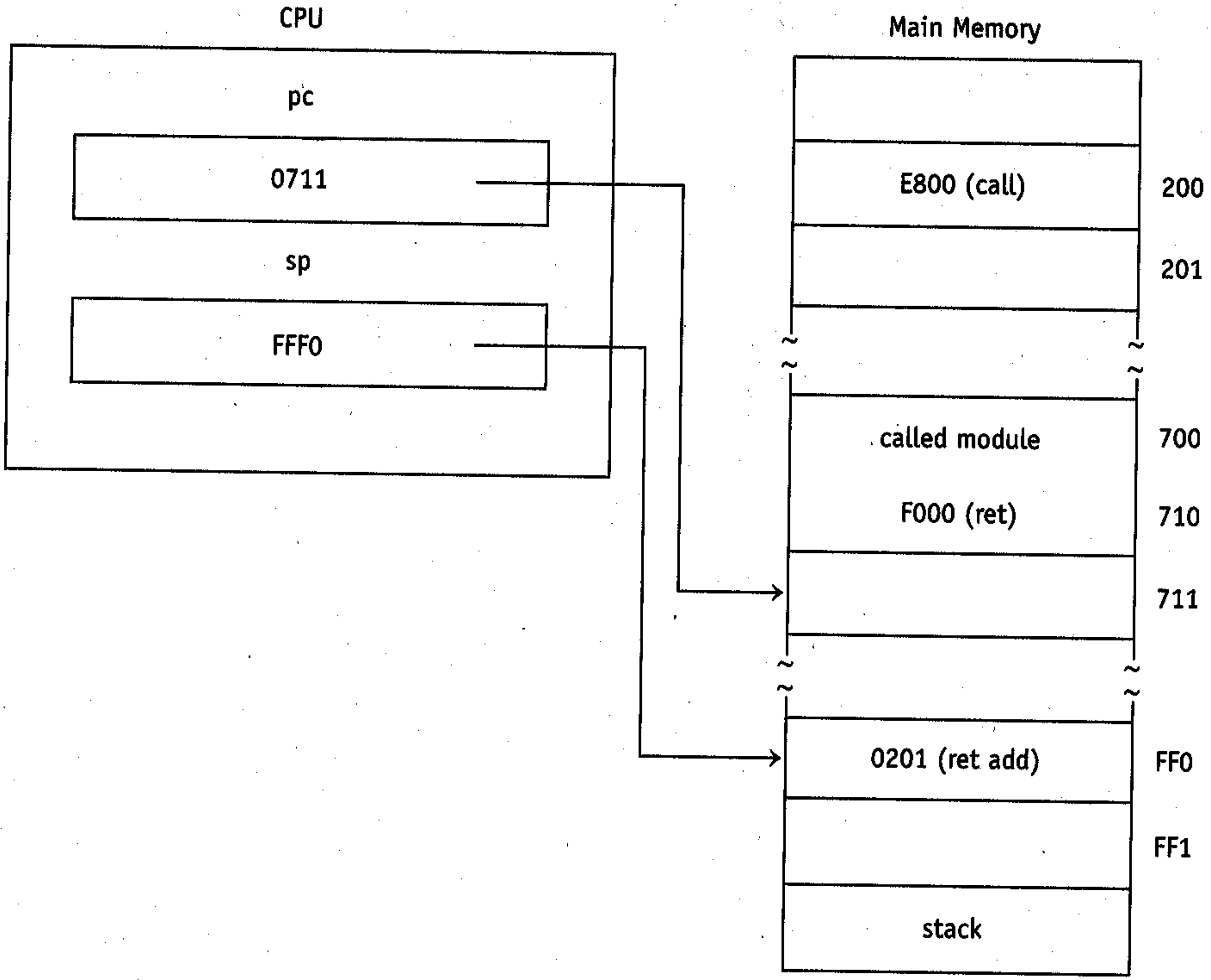


b) Execution of call instruction has been completed. The return address has been pushed and a jump to called module has occurred.

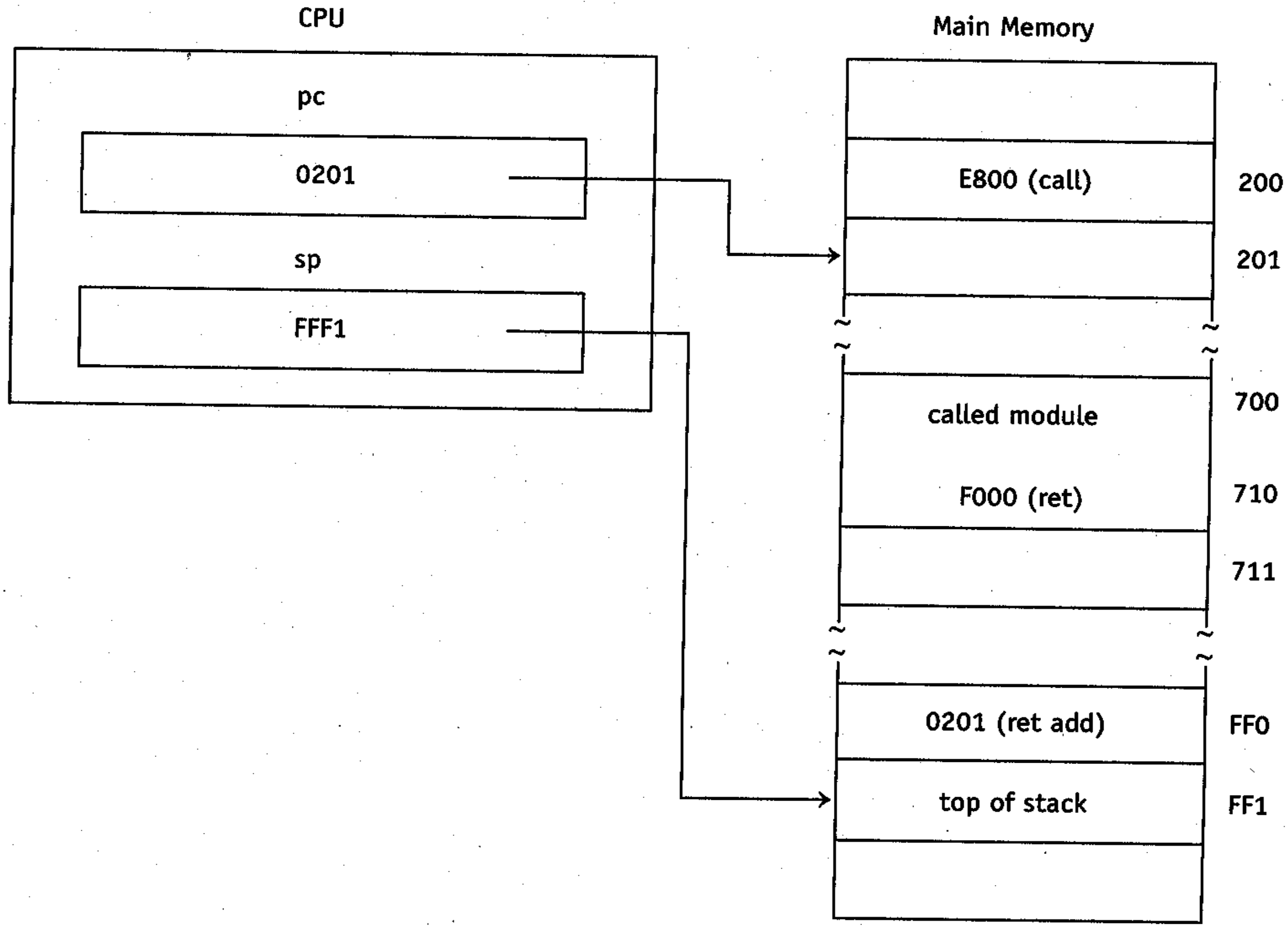


*(continued)*

**FIGURE 4.21** (continued) c) About to execute ret instruction (step 4 of CPU cycle).



d) Execution of the ret instruction has been completed. The return address has been popped into the pc register. The instruction at the return address is about to be fetched and executed.



The program on the next slide has three modules: a **main** module that calls **f1** and **f2**.

FIGURE 4.22

```
         ;          main module -- illustrates call instruction
         call f1
ret1:    ldc  msgmain
         sout
         call f2
ret2:    halt
msgmain: dw   "middle"
; ==========================================================
         ;          module f1 -- outputs string "left"
f1:      ldc  msgf1
         sout
         ret
msgf1:   dw   "left"
; ==========================================================
         ;          module f2 -- outputs string "right\n"
f2:      ldc  msgf2
         sout
         ret
msgf2:   dw   "right\n"
```

Can we  replace call/ret with ja instructions?

Yes, but what about the next slide?

FIGURE 4.20

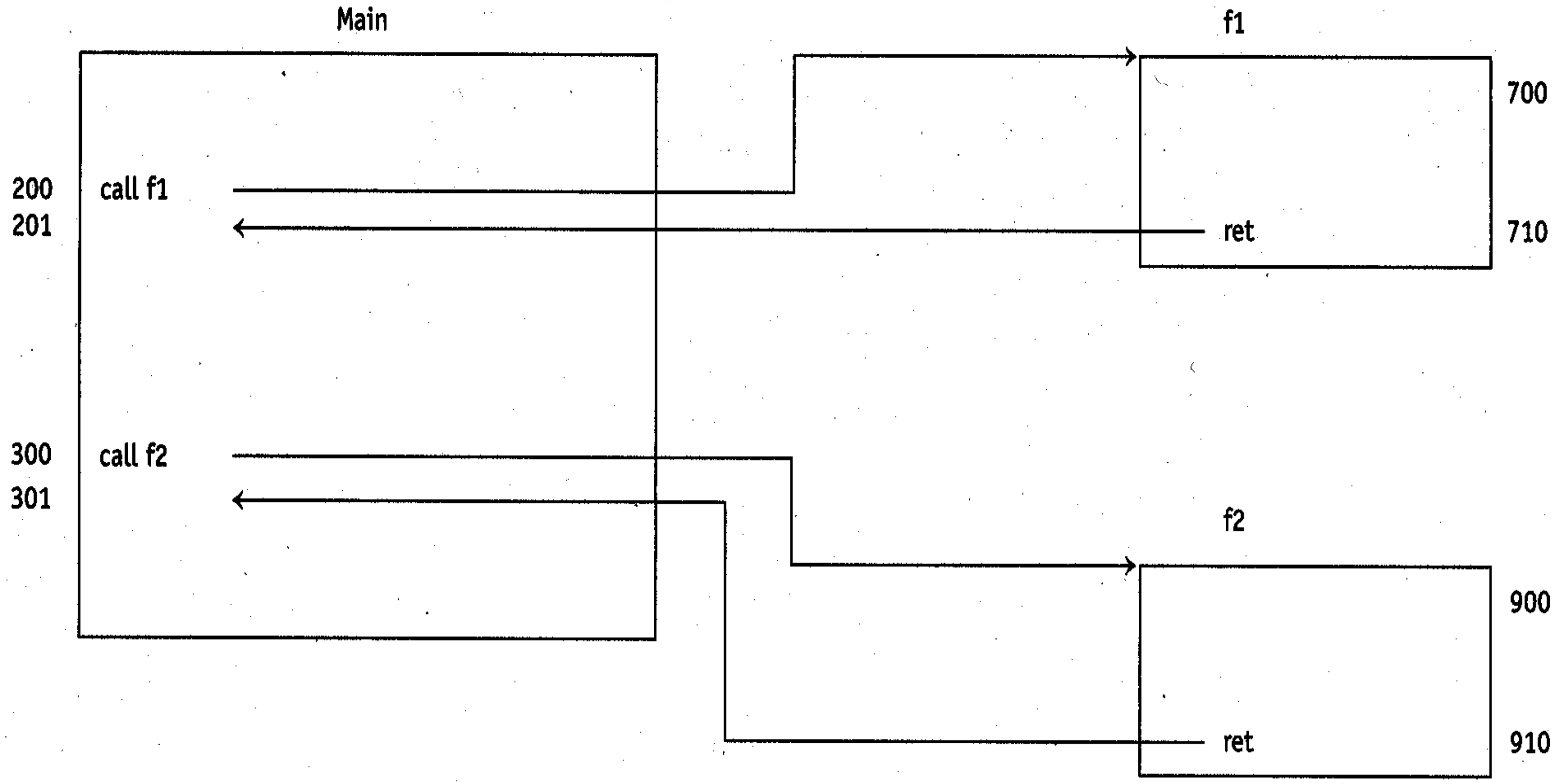
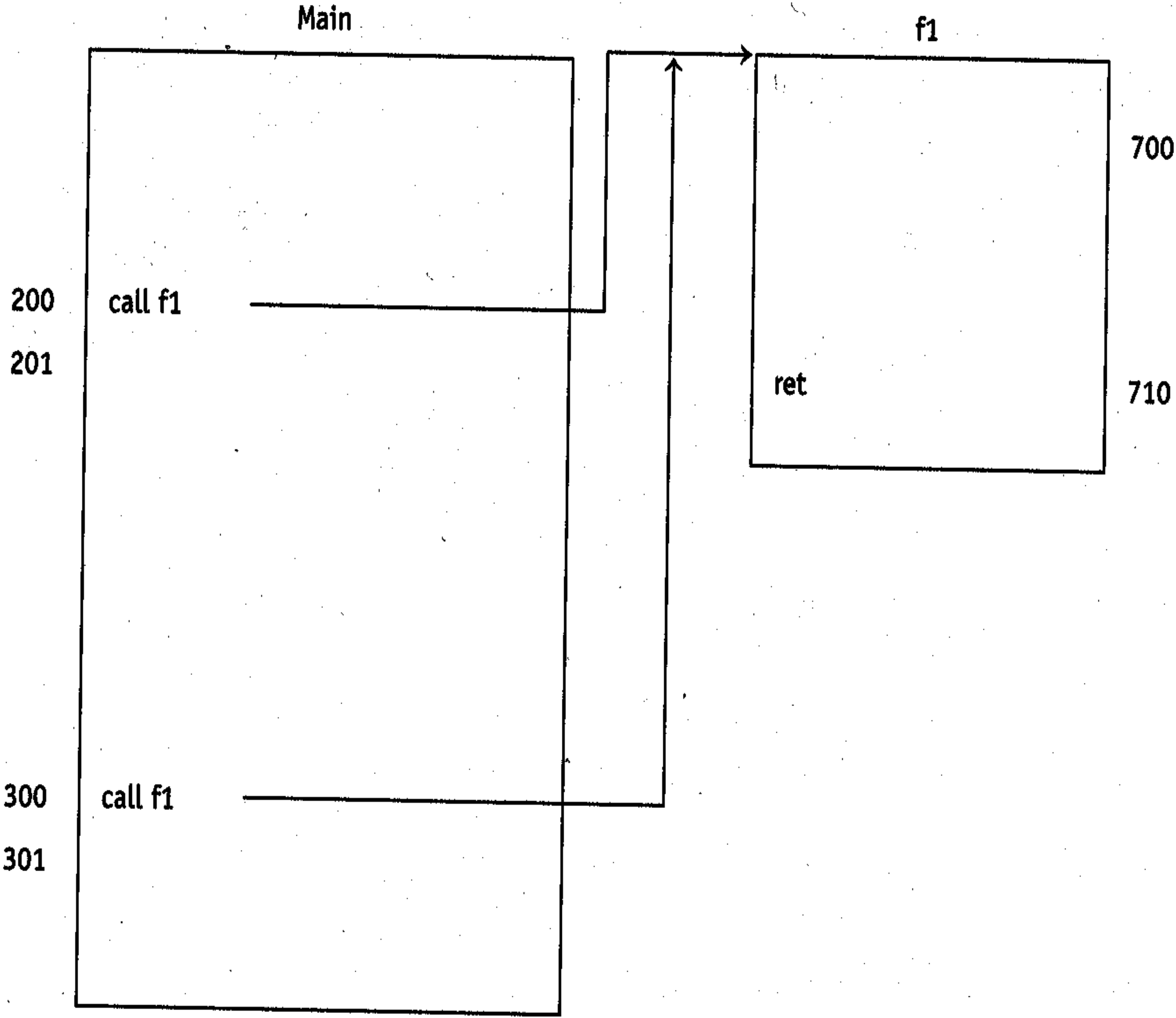


FIGURE 4.23



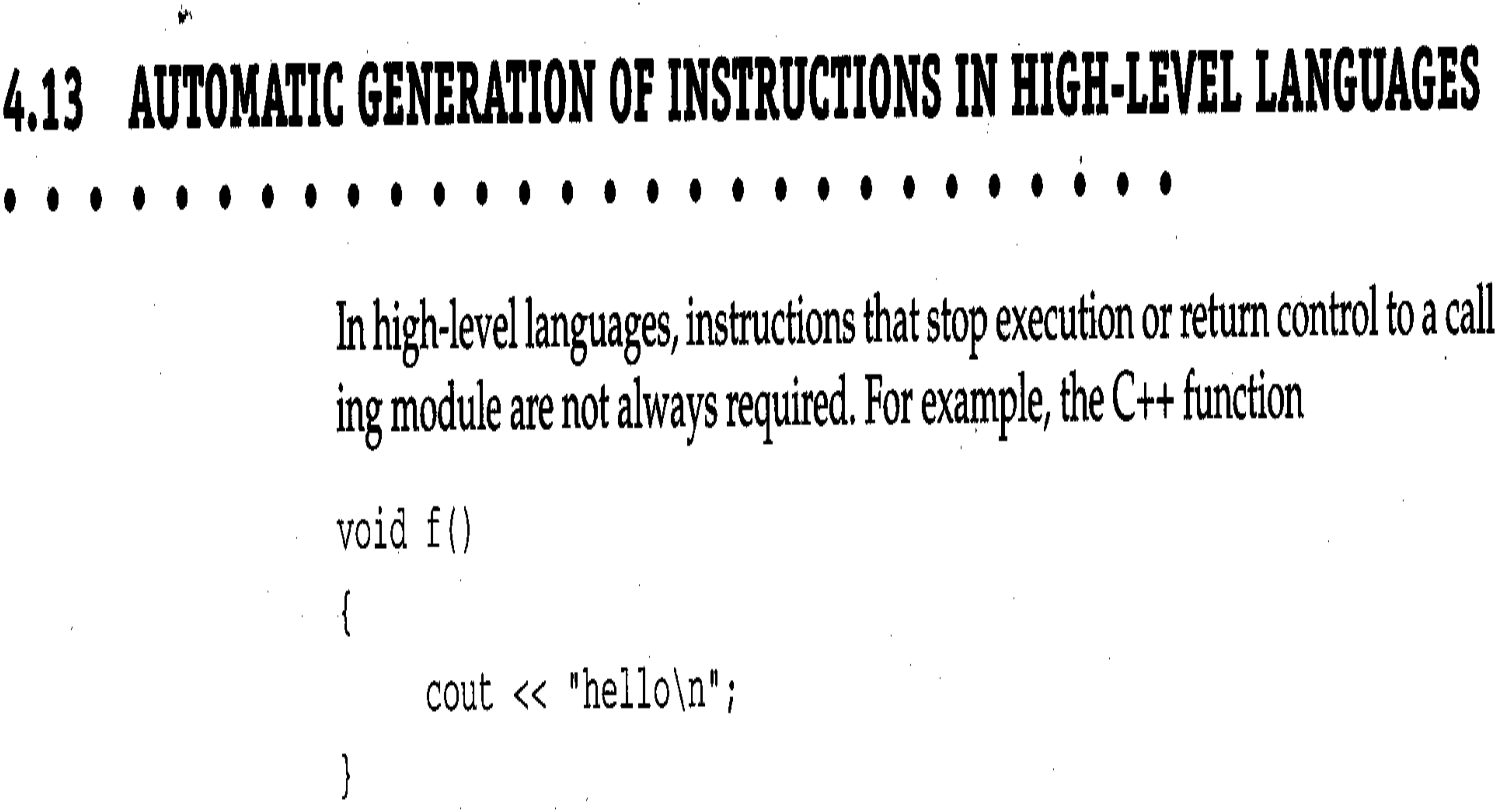
## 4.12 TERMINATING INSTRUCTIONS

| Opcode (hex) | Assembly Form | Name | Description |
|---|---|---|---|
| FFFE | bkpt | Breakpoint | Trigger breakpoint |
| FFFF | halt | Halt | Trigger halt |

# Terminating instructions

- Halts terminates program.  Must restart from beginning to continue (by entering o, then g or t).
- Bkpt stops execution—can restart from the current execution point (by entering the  t or g commands).  Used for debugging.
- 16-bit opcode

# Assembler does not automatically generate instructions.



## 4.13 AUTOMATIC GENERATION OF INSTRUCTIONS IN HIGH-LEVEL LANGUAGES

In high-level languages, instructions that stop execution or return control to a calling module are not always required. For example, the C++ function

```
void f()
{
    cout << "hello\n";
}
```

# Some debugging commands

- b12 sets breakpoint at location 12 hex
- k (or b-) kills breakpoint
- w20 sets watchpoint at location 20 hex
- kw (or w-) kills watchpoint
- mr sets “plus reads” mode
- ms sets “plus source” mode
- mso sets “plus source only” mode
- mr- cancels “plus reads”
- ms- cancels source modes

# Plus source mode

```
---- [T7] 0: ld  /0 018/ ms
Machine-level display mode + source
---- [T7] 0: ld  /0 018/ g
  0: loop:     ld       sum         ; get current sum
     ld        /0 018/  ac=0000/0000
  1:           add    n             ; add n to it
     add       /2 010/  ac=0000/0014

               .
               .
               .
```

# Source-only mode

```
---- [T7]  0:  ld    /0 018/  mso
Machine-level display mode source only
---- [T7]  0:  ld    /0 018/ g
  0: loop:     ld      sum          ; get current sum
  1:           add   n              ; add n to it
          .
          .
          .
```

# Plus reads mode

```
---- [T7]  0:  ld    /0 018/  mr
```

Machine-level display mode + reads

```
---- [T7]  0:  ld    /0 018/  g
  0: ld    /0 018/  0000<m[018]  ac=0000/0000
  1: add /2 010/  0014<m[010]  ac=0000/0014
        .
        .
        .
```

Watchpoint: execution stops when contents of the specified location changes.

Breakpoint: execution stops when instruction at the specified location is about to be executed.

May use labels to specify a location when setting a watchpoint or breakpoint if sim is in the plus source or source only mode.  Otherwise, you must use absolute addresses.

| | |
|---|---|
| wn | (set watchpoint at ‘n’) |
| bloop | (set breakpoint a ‘loop’) |
| w10 | (set watchpoint a location 010) |
| b0 | (se breakpoint at location 000) |

# Watchpoint

```
---- [T7]  0:  ld    /0 018/ wn
Watchpoint set at ‘n’ (loc 10)
---- [T7]  0:  ld    /0 018/  g  (execute until n changes)
     . . .
     . . .
Watchpoint at ‘n’ (loc 10)    m[010] = 11
---- [T7] 5:  jz    /C 008/
```

# Breakpoint

```
---- [T7]  0:  ld    /0 018/ bloop
Breakpoint set at ‘loop’ (loc 20)
---- [T7]  0:  ld    /0 018/  g  (execute  )
     . . .
     . . .
Breakpoint at ‘loop’ (loc 20)
---- [T7]  20: sub   /0 00F/
```

Cancel watchpoint with w- or kw

Cancel breakpoint with b- or k

# More debugging commands

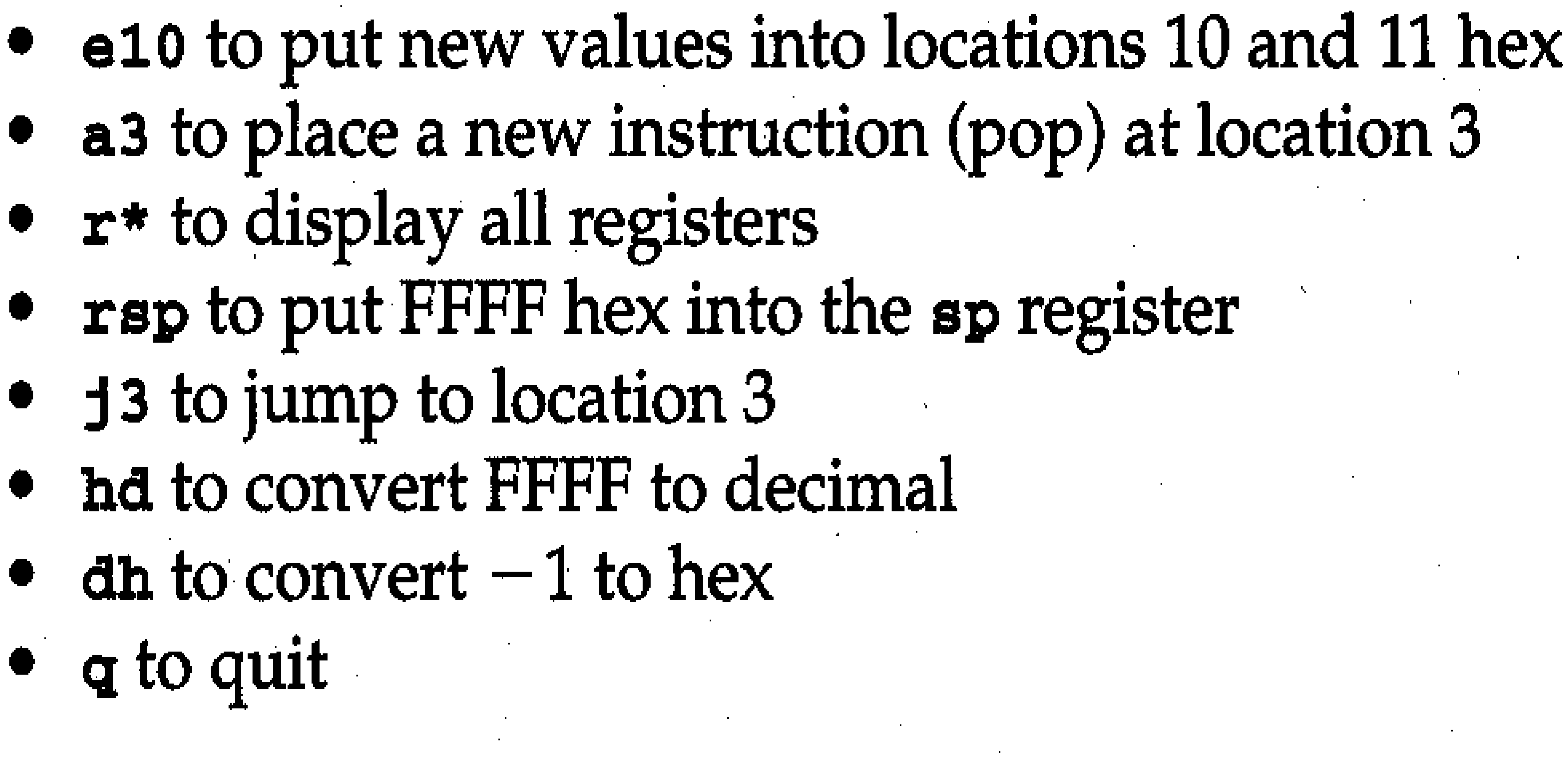
- d100 display starting at location 100 hex
- d$ display stack
- d@ display memory pointed to by ac
- d* display all
- u% unassemble instructions to be executed next.
- p triggers partial trace

The p command is useful for detecting the location of an infinite loop.

**FIGURE 4.24**

```
Starting session.     Enter h or ? for help.

— [T7] 0: ldc  /8 005/ p

Partial machine-level display mode

— [T77]t7

0/1/2/3/4/5/6/

— [T7]                    ←hit ENTER to invoke default command T7

3/4/5/6/3/4/5/

— [T7]                    ←hit ENTER to invoke default command T7

6/3/4/5/6/3/4/

— [T7]                    ←hit ENTER to invoke default command T7

5/6/3/4/5/6/3/

— [T7] q
```

# More debugging commands



- `e10` to put new values into locations 10 and 11 hex
- `a3` to place a new instruction (pop) at location 3
- `r*` to display all registers
- `rsp` to put FFFF hex into the `sp` register
- `j3` to jump to location 3
- `hd` to convert FFFF to decimal
- `dh` to convert −1 to hex
- `q` to quit

FIGURE 4.25

```
Starting session.     Enter h or ? for help.
---- [T7] 0: ld  /0 018/ e10                 ←edit from loc 10 hex
 10:  0014/003a                              ←enter new value for loc 10
 11:  0053/003b                              ←enter new value for loc 11
 12:  0075/                                  ←hit ENTER to exit edit mode
---- [T7] 0: ld  /0 018/ a3                  ←assemble from loc 3
  3:  ld   /0 010/ pop                       ←assemble new inst for loc 3
  4:  sub  /3 00F/                           ←hit ENTER to exit assembly mod
---- [T7] 0: ld  /0 018/ r*                  ←display all registers
      pc    = 0000    sp    = 0000     ac    = 0000
---- [T7] 0: ld  /0 018/ rsp                 ←display/edit sp
      sp = 0000/ffff                         ←enter a new value for sp
---- [T7] 0: ld  /0 018/ j3                  ←jump to loc 3
---- [T7] 3: pop /F4 00/ hd ffff             ←convert hex to decimal
      unsigned: 65535 (dec)  signed: -1 (dec)
---- [T7] 3: pop /F4 00/ dh -1               ←convert decimal to hex
      FFFF (hex)
---- [T7] 3: pop /F4 00/ q                   ←quit
```

With the debugger, you can see how instructions work.  The next slide shows how the push instruction works.

**FIGURE 4.26**

```
C:\H1>sim none
Simulator Version x.x

Starting session.  Enter h or ? for help.
---- [T7]  0: ld   /0 000/ rac
     ac = 0000/5                      ←set ac to 5
---- [T7]  0: ld   /0 000/ rsp
     sp = 0000/ffff                   ←set sp to ffff
---- [T7]  0: ld   /0 000/ efff       ←edit memory starting at loc fff
FFF:  0000/7
Now at upper limit of main memory--exiting edit mode
---- [T7]  0: ld   /0 000/ a0         ←assemble starting at loc 0
  0:  ld   /0 000/ push
  1:  ld   /0 000/                    ←hit ENTER to exit assembly mode
---- [T7]  0: push /F1 00/ t1         ←execute one instruction
  0:  push /F3 00/ m[FFE]=0000/0005 sp=FFFF/FFFE
---- [T1]  1: ld   /0 000/ r*         ←display all registers
     pc    = 0001    sp =    FFFE    ac    = 0005
---- [T1]  1: ld   /0 000/ d$         ←display stack
FFE:  0005 0007
---- [T1]  1: ld   /0 000/ q          ←quit
```

Using the debugger, it is easy to find the errors in the program on the next slide.  This program calls a module that adds two numbers and returns the sum in the ac register.

FIGURE 4.27

```
get_sum: ldr  0         ; get second number
       addr 1           ; add first number
       ret              ; return sum in ac register
;=================================================
main:  ld    x
       push             ; push first number
       ld    y
       push             ; push second number
       call get_sum     ; call function which adds two numbers
       dout             ; display number
       ldc   '\n'       ; output newline character
       aout
       halt
       x:    dw         2
       y:    dw         3
```

**FIGURE 4.28** a)

```
Starting session. Enter h or ? for help.
---- [T7] 0: ldr       /4 000/                      ←hit ENTER to invoke T7
  0:  ldr   /4 000/   ac=0000/4000                  ←wrong entry point
  1:  addr  /6 001/   ac=4000/A001
  2:  ret   /F0 00/   pc=0003/4000  sp=0000/0001
  0:  ldr   /4 000/   ac=A001/6001
  1:  addr  /6 001/   ac=6001/5001
  2:  ret   /F0 00/   pc=4003/6001  sp=0001/0002
  1:  addr  /6 001/   ac=5001/500D
---- [T7] 2: ret  /F0 00/
```

The entry point is wrong.  Use the j debugger command to start from the correct instruction (you can fix to source code later).

b)

```
---- [T7] 2: ret  /F0 00/ o                ←do over
Starting session.  Enter h or ? for help.
---- [T7] 0: ldr      /4 000/ j3           ←jump to correct entry point
---- [T7] 3: ld       /0 00C/ g            ←go to halt
  3:  ld   /0 00C/   ac=0000/0002
  4:  push /F3 00/   m[FFF]=0000/0002  sp=0000/FFFF
  5:  ld   /0 00D/   ac=0002/0003
  6:  push /F3 00/   m[FFE]=0000/0003  sp=FFFF/FFFE
  7:  call /E 000/   m[FFD]=0000/0008  pc=0008/0000  sp=FFFE/FFFD
  0:  ldr  /4 000/   ac=0003/0008          ←this load not working correctly
  1:  addr /6 001/   ac=0008/000B
  2:  ret  /F0 00/   pc=0003/0008  sp=FFFD/FFFE
  8:  dout /FFFD /   11                    ←wrong answer
  9:  ldc  /8 00A/   ac=000B/000A
  A:  aout /FFFB /

  B:halt /FFFF /
 Machine inst count =    C (hex) =    12 (dec)
 ---- [T7]
```

The ldr instruction is using the wrong relative address.  Use the **a** command to assemble a new ldr instruction.  Use o# so that the modification is not overlaid.

```
---- [T7] a0                                  ←assembly from loc 0
  0: ldr  /4 000/   ldr 1                     ←correct relative address
  1: addr /6 001/   addr 2                    ←correct relative address
  2: ret  /F0 00/
---- [T7] o#                                  ←do over but don't reinit mem
Starting session. Enter h or ? for help.
---- [T7] 0: ldr      /4 001/ j3              ←jump to correct entry point
---- [T7] 3: ld       /0 00C/ g               ←go to halt
  3: ld   /0 00C/   ac=0000/0002
  4: push /F3 00/   m[FFF]=0002/0002  sp=0000/FFFF
  5: ld   /0 00D/   ac=0002/0003
```

*(continued)*

FIGURE 4.28 (continued)

```
6: push /F3 00/   m[FFE]=0003/0003  sp=FFFF/FFFE
7: call /E 000/   m[FFD]=0008/0008  pc=0008/0000  sp=FFFE/FFFD
0: ldr  /4 001/   ac=0003/0003
1: addr /6 002/   ac=0003/0005
2: ret  /F0 00/   pc=0003/0008  sp=FFFD/FFFE
8: dout /FFFD /   5                           ←correct answer
9: ldc  /8 00A/   ac=0005/000A
A: aout /FFFB /


B: halt /FFFF /
Machine inst count =    C (hex) =    12 (dec)
---0 [T7] q
```

# Memory-mapped I/O

- Associates an I/O device with a specific set of absolute addresses.
- Load and store operations to these addresses trigger I/O operations.
- Put ‘&’ at the beginning of an assembly language program to activate memory-mapped I/O.

**FIGURE 4.29** Memory-Mapped Locations

| Location (decimal) | Function |
|---|---|
| 3000 | Keyboard status (1 = ready; 0 = not ready) |
| 3001 | Keyboard data |
| 3002 | Monitor status (1 = ready; 0 = not ready) |
| 3003 | Monitor data |

**FIGURE 4.30**

a)

```
&                   ; configure H1 for memory-mapped I/O
ld      3002        ; get status word from display monitor
jz      * - 1       ; if 0 (not ready), try again
ldc     'A'         ; get 'A'
st      3003        ; store in data word for display monitor
halt
```

b)

```
---- [T7] 0: ld   /0 F3A/ g
  0: ld      /0 BBA/    ac=0000/0001          ←1 indicates monitor is ready
  1: jz      /C 000/                          ←no jump because status = 1
  2: ldc     /8 041/    ac=0001/0041          ←get 'A'
  3: st      /1 BBB/    m[BBB]=0000/0041      ←output the 'A' to monitor
  4: halt    /FFFF /    A                     ←'A' displayed after a delay
Machine inst count =   5 (hex) =     5 (dec)
---- [T7] q
```

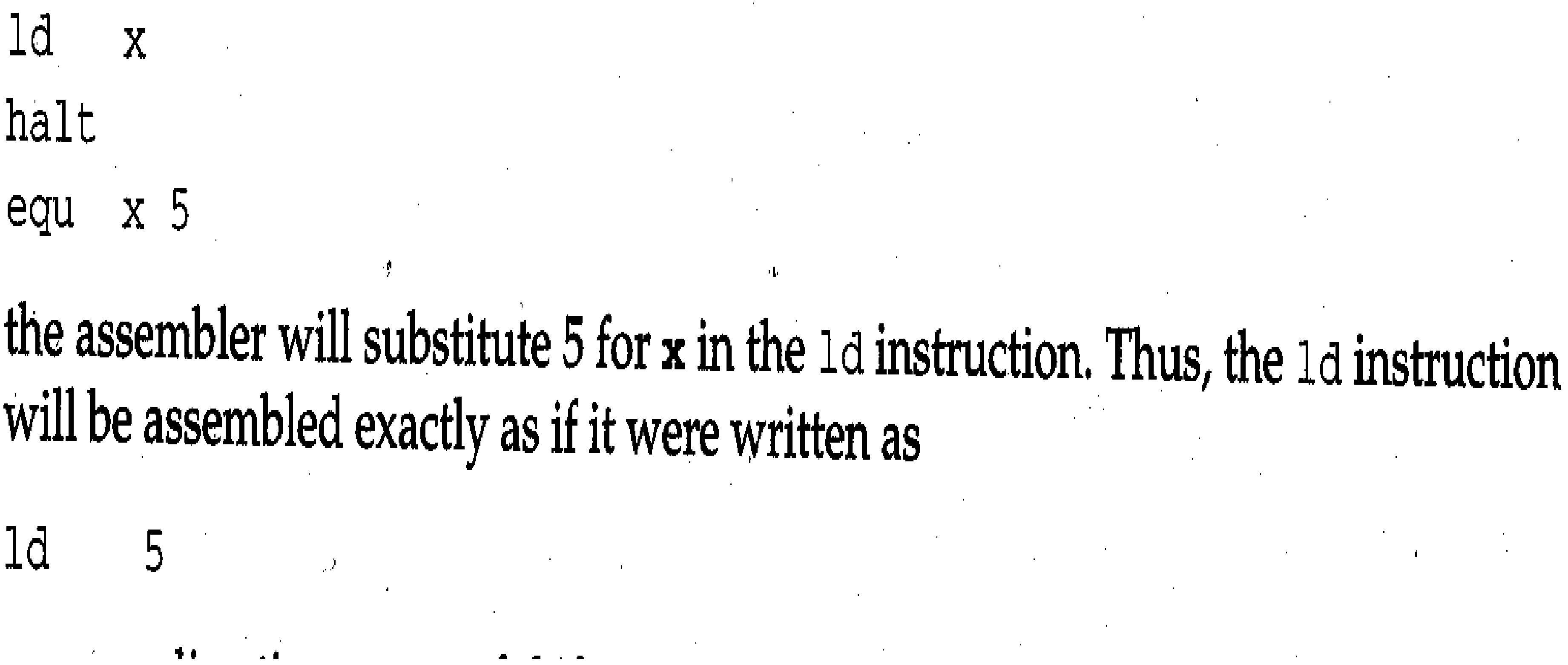
FIGURE 4.31 a)

```
&                  ; configure H1 for memory-mapped I/O
ld    3000         ; get keyboard status
jz    * -1         ; jump if not ready
ld    3001         ; get character from keyboard
halt
```

b)

```
---- [T7] 0: ld   /0 BB8/ g
   0: ld      /0 BB8/   ac=0000/0000  A        ←user enters 'A'
   1: jz      /C 000/   pc=0002/0000
   0: ld      /0 BB8/   ac=0000/0001           ←char now available
   1: jz      /C 000/                          ←no jump because status = 1
   2: ld      /0 BB9/   ac=0001/0041           ←read char from keyboard
   3: halt    /FFFF /
Machine inst count =   6 (hex) =    6 (dec)
---- [T7] q
```

# equ directive



```
ld    x
halt
equ   x 5
```

the assembler will substitute 5 for **x** in the `ld` instruction. Thus, the `ld` instruction will be assembled exactly as if it were written as

```
ld    5
```

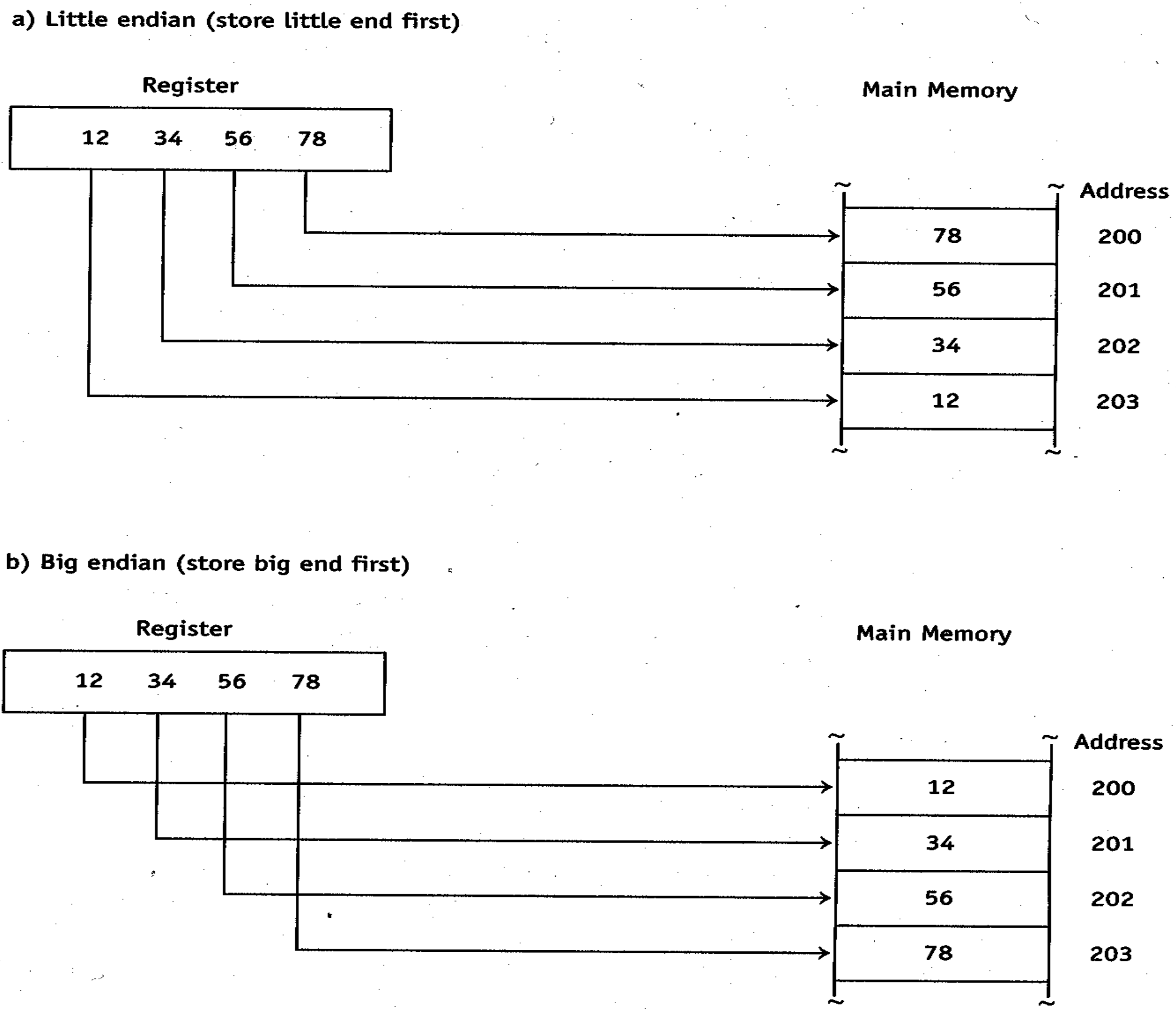
FIGURE 4.32

```
&                       ; configure H1 for memory-mapped I/O
ld     kbstatus         ; get keyboard status
jz     * -1             ; jump if not ready
ld     kbdata           ; get character from keyboard
halt
equ    kbstatus 3000
equ    kbdata   3001
```

# Little endian or big endian

Depends on how multiple-byte words are stored in byte-addressable memory

FIGURE 4.33



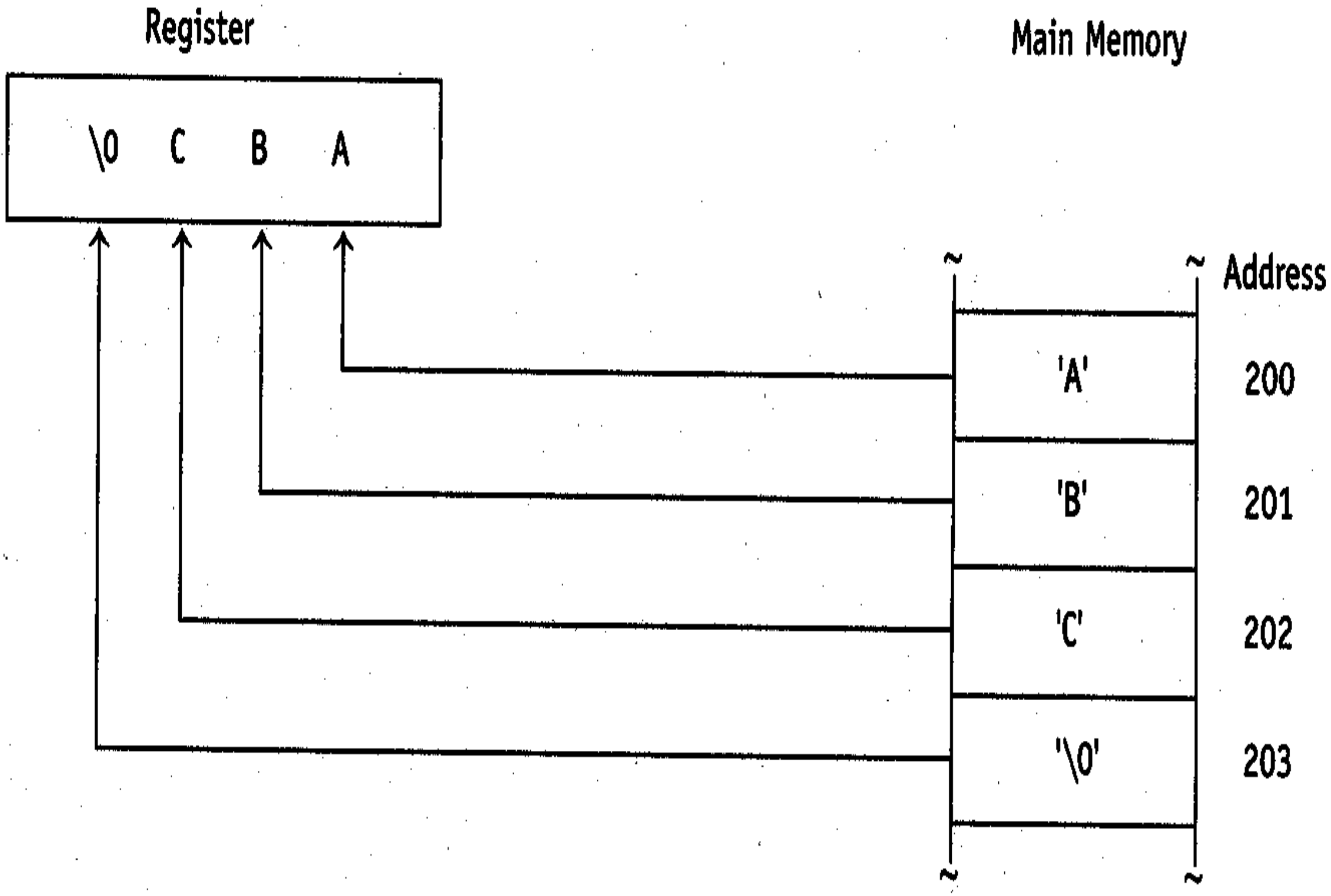
a) Little endian (store little end first)

b) Big endian (store big end first)

Little endian not good for string comparisons that hold the strings in registers—collating order is wrong.

See the next slide (‘C’ on the next slide is more significant than ‘A’).

**FIGURE 4.34** Loading a string on a little-endian computer



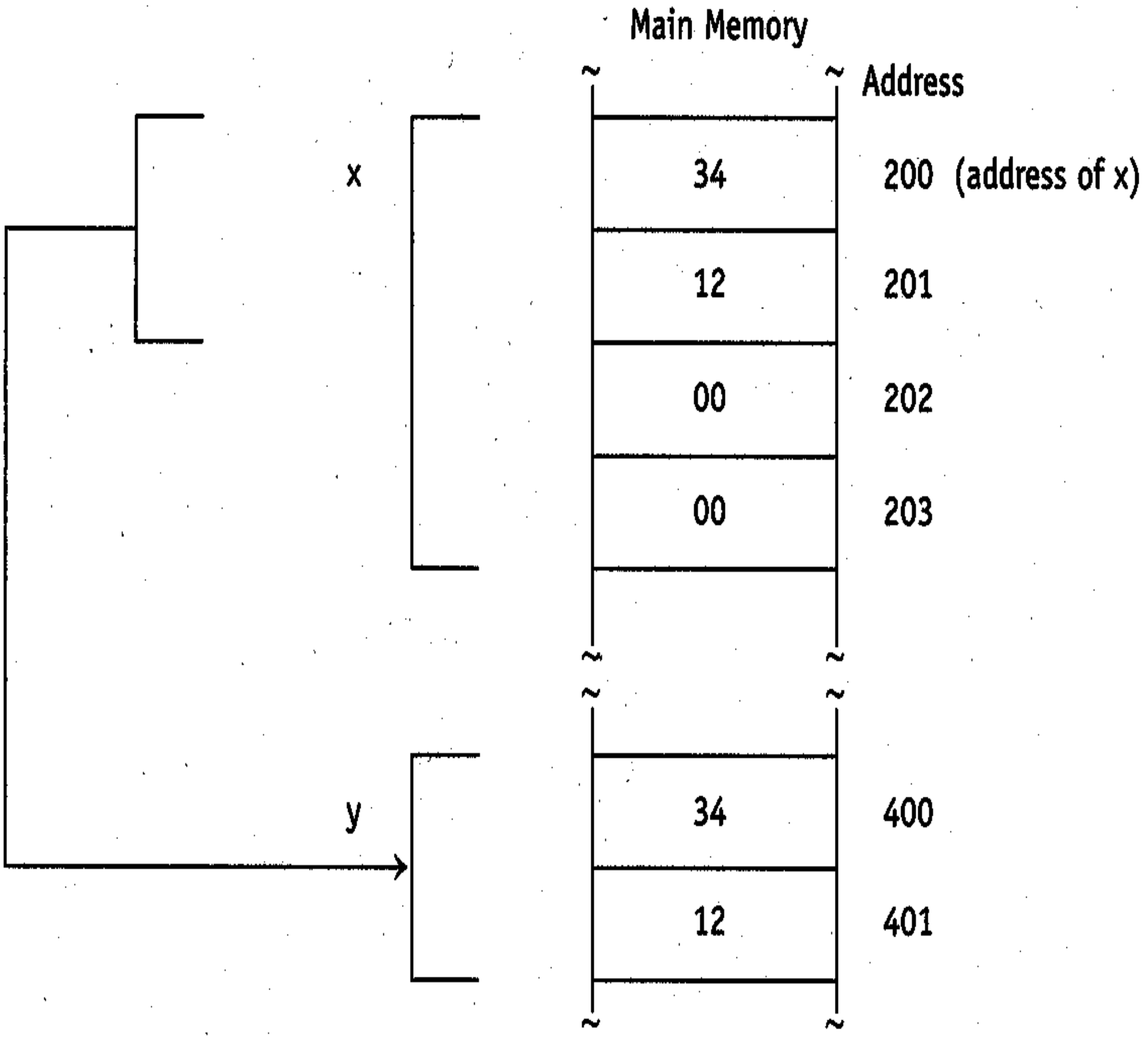
# Assume int is 4 bytes; short, 2 bytes. Then little endian is better for

```
int x = 0x00001234;
short y;
y = (short)x;
```

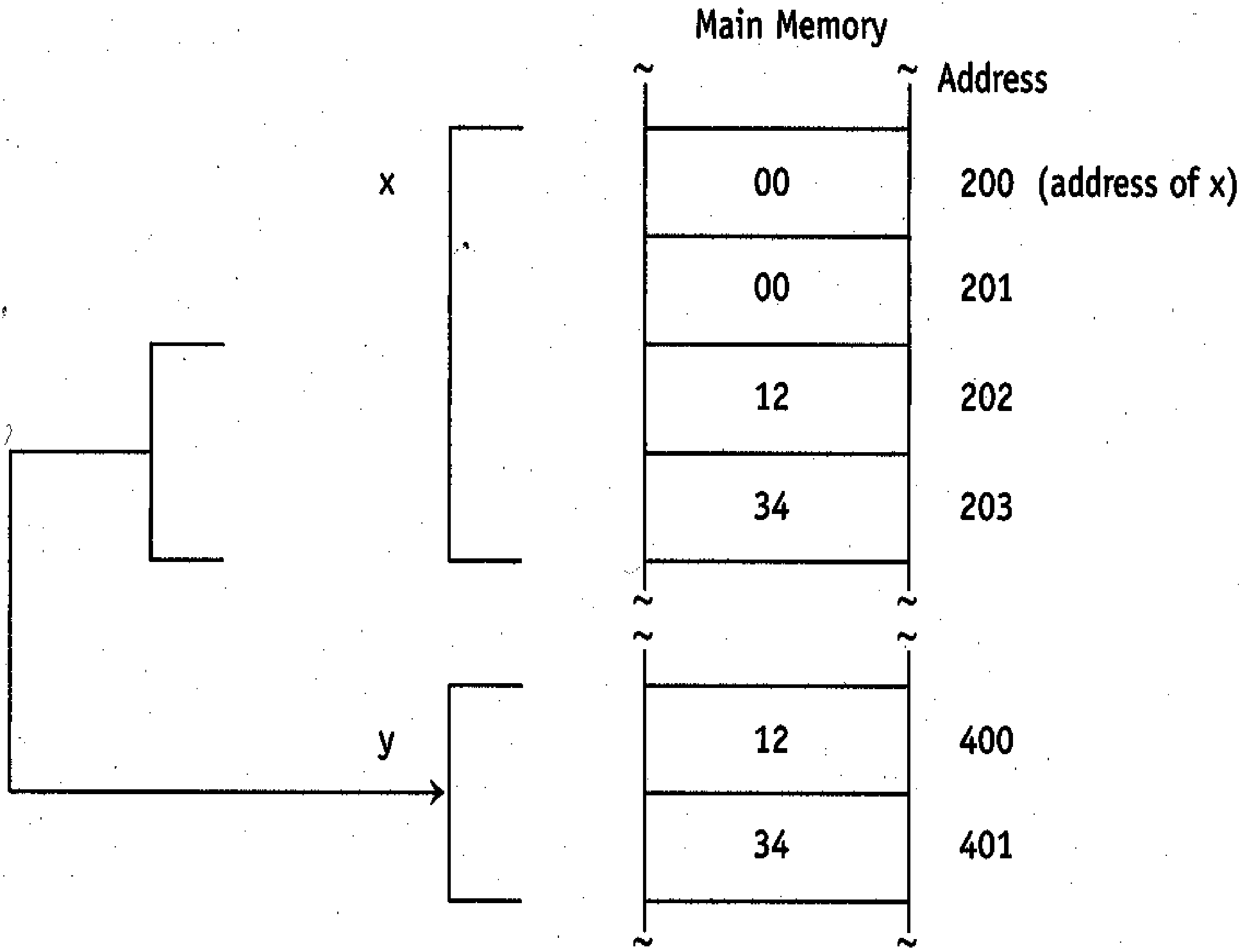
because you do not have to compute the location of the less significant half of x.

See the next slide.

**FIGURE 4.35** a) Precision conversion on a little-endian computer



b) Precision conversion on a big-endian computer


Main Memory
Address
x
00
200 (address of x)
00
201
12
202
34
203
y
12
400
34
401